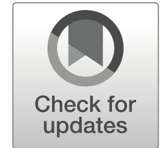
Peer-to-Peer Networking and Applications (2020) 13:439–474
https://doi.org/10.1007/s12083-019-00782-8



New efficient M2C and M2M mutual authentication protocols for IoT-based healthcare applications

Fatma Merabet[1,2] · Amina Cherif[1,2] · Malika Belkadi[1] · Olivier Blazy[2] · Emmanuel Conchon[2] · Damien Sauveron[2]

Received: 12 March 2019 / Accepted: 21 June 2019 / Published online: 2 August 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
With the rapid advancement of heterogeneous wireless technologies and their proliferation in ambient connected objects, the Internet of Things (IoT) is a paradigm that revolutionizes communication between people/objects. Communication between connected objects is achieved via various communication modes, including Machine-to-Machine (M2M) and Machine-to-Cloud (M2C). In the medical field, monitoring devices help to collect, exchange and process patient health parameters, and are employed in open and unprotected environments, which expose them to various attacks. For this reason, providing high levels of security and privacy become crucial, and a first requirement to ensure this is authentication. In this paper, we propose three new lightweight, efficient authentication protocols for IoT-based healthcare applications. We formally verify them using AVISPA and ProVerif automated tools. For each protocol, we provide a security analysis and a performance evaluation that we compare to related existing proposals.

Keywords IoT · M2C · M2M · Authentication protocol · Healthcare

1 Introduction

Connected objects are increasingly invading our everyday life. These intelligent objects are equipped with sensing capacities (temperature, pressure, vibration, luminosity, humidity, voltage, etc.) and are able to communicate through various communication technologies (including RFID, NFC, Bluetooth, Wi-Fi and LoRa). Thanks to these communication technologies, smart objects can interact in real time, cooperate and anticipate some actions to achieve common goals (such as environmental monitoring, urban traffic control, and responses to natural disasters.). Together, these smart objects are the base of the Internet of Things (IoT) paradigm [1], which allows people and objects to be connected anytime and anywhere [2]. The IoT involves the inclusion of smart devices into almost any physical objects to form smart objects (hereafter referred to as connected devices or connected objects) to enable them communicate and share with each other via the network and to collect data from any location in the world. The IoT is characterized by heterogeneous technologies that collaborate to provide innovative services in various fields of application [3, 4]. It aims to provide advanced communication between smart objects and systems to facilitate human interaction with the virtual world. Communication between connected objects is achieved with various types of communication modes including Machine-to-Machine (M2M) and Machine-to-Cloud (M2C). In an M2M communication mode, smart objects can exchange and share data in a decentralized manner, without involving a centralized system [5–7]. In contrast, M2C communication mode provides centralized communication between smart objects and the cloud.

The IoT has numerous application domains including smart transport, smart home, smart cities, and smart healthcare. In the medical field, patients use smart objects that offer medical services the possibility of diagnosing and determining the causes of certain pathologies. The proliferation of smart objects through home automation solutions can also bring many advantages to people with reduced mobility, such as the elderly or disabled people, by allowing them greater autonomy and improving their well-being in the context of Ambient Assisted Living

This article is part of the Topical Collection: *Special Issue on IoT System Technologies based on Quality of Experience*
Guest Editors: Cho Jaeik, Naveen Chilamkurti, and SJ Wang

✉ Damien Sauveron
damien.sauveron@unilim.fr

Extended author information available on the last page of the article.

(AAL) [8]. To convince users to adopt IoT-based solutions, security is a key challenge to address. Therefore, to secure communication between different objects, security protocols must be proposed. Generally, smart objects have energy, memory and computational constraints, which make the use of traditional security protocols unsuitable and require the development of new and appropriate security protocols.

To counter threats and attacks to which IoT infrastructure communications are exposed, a set of security services should be guaranteed. An authentication service ensures, through authentication protocols, that communicating entities are who they claim to be. In the IoT context, authentication protocols are often based on lighter cryptographic algorithms adapted to constrained devices, such as Elliptic Curve Cryptography (ECC). This authentication phase is often a prerequisite mandatory step to enable parties to securely communicate; therefore the aim of this paper is to propose efficient authentication protocols that can be applied in various IoT-based healthcare applications.

1.1 Contribution

The main purpose of this paper is to propose efficient M2C and M2M mutual authentication protocols and to compare their security and performance with several other existing protocols for IoT-based healthcare applications.

The salient contributions of this paper are as follows:

1. Adding two main improvements (at both sides) of a recent M2C mutual authentication protocol proposal for a healthcare RFID system.
2. Proposing a new, more efficient and scalable M2C mutual authentication protocol, which is mainly hash-based.
3. Proposing a new M2M mutual authentication protocol using ECC.
4. Validating the proposed protocols with AVISPA and ProVerif automated formal verification tools and conducting performance evaluations on IoT devices.

1.2 Structure of the paper

Section 2 presents an overview of an IoT-based architecture for healthcare applications and the related security and functional requirements. Section 3 provides a brief summary of related work on authentication protocols in M2C and M2M communication modes from a general IoT perspective. In section 4, two M2C and one M2M authentication protocols that fulfill our requirements are proposed and for each an informal security analysis is provided, complemented with formal analysis using AVISPA and ProVerif automated verification tools. A comparative analysis of the features and performances of these proposals with the current state-of-the-art is provided in Section 5. Section 6 concludes the paper.

2 Overview of an IoT-based architecture for healthcare applications

By definition, healthcare has different areas, including disease prevention, health maintenance and restoration, and taking care of patients' overall well-being. To illustrate how an IoT-based architecture can be helpful for various healthcare applications, Fig. 1 depicts a smart home scenario using the two communication modes (M2M and M2C).

Smart homes offer users an increased level of independence and improve their quality of life. Generally speaking, a Home Area Network (HAN) consists of various smart objects such as smart lights, smart clocks, and temperature sensors. These objects can communicate directly without human intervention in various ways. For instance, a light sensor can detect the absence of sunlight and send a message to switch on a smart light without any human intervention, in a M2M communication mode. In an M2C communication mode, a smart thermostat can connect to the weather forecast service available on the cloud to determine whether it needs to modify the temperature (by sending M2M requests to relevant appliances such as heaters or air conditioning units) as requested by the user.

In the healthcare example illustrated in Fig. 1, several smart objects are present in the smart home: a) medical devices worn by the patient; b) environmental devices (sensors and actuators); and c) appliances and multimedia devices. They collaborate and communicate in different ways. For instance, wearable devices on the human body form a Body Area Network (BAN) and they are generally connected using an M2M communication mode to a single access point, which can be an external access point such as a home gateway, or a Body Central Unit (BCU), a patient-carried device to collect the patient's data from the various body sensors. Then, the data are transmitted either to a local management device such as the patient's phone or directly to remote servers in the cloud. The M2C communication mode is used to enable medical officers to monitor the health status of the patient and retrieve his/her medical records as needed.

2.1 Security requirements

Because the communication channels between two smart objects (M2M) or a smart object and a remote server in the cloud (M2C) are prone to various attacks, and because medical applications require that data related to patients'

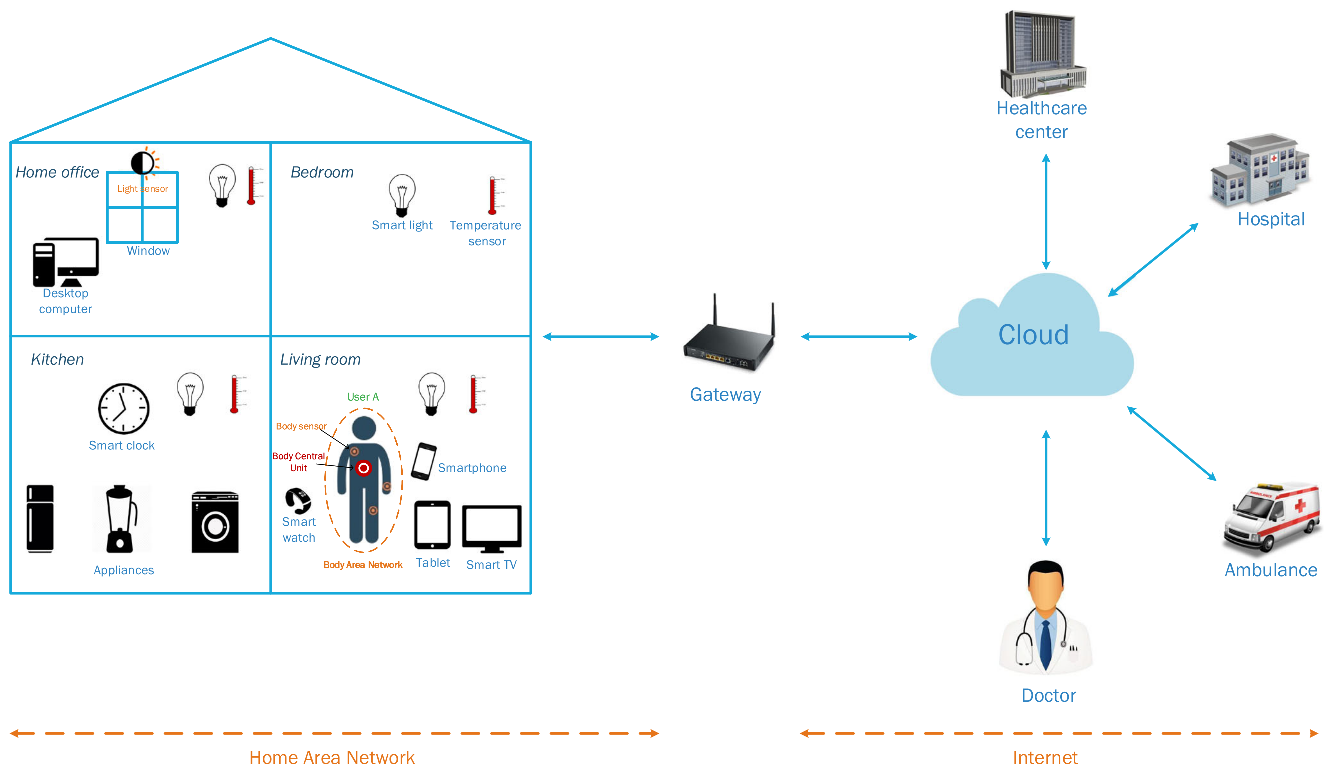


Fig. 1 Healthcare applications in a smart home scenario

health must be handled to ensure their confidentiality, integrity and privacy, authentication is a requirement in order to establish session keys that will be used in future communications to protect data exchanges.

- (SR1) Authentication: Two communicating entities (smart objects in M2M communication mode or a smart object and the cloud in M2C communication mode) should mutually authenticate themselves before any further interaction. Each party should provide a proof of its identity.
- (SR2) Confidentiality, integrity and privacy: To ensure the confidentiality, integrity and privacy of exchanged data between communicating parties, a fresh session key is required.

2.2 Functional requirements

- (FR1) Efficiency:
  - (FR1a) Authentication Processing Time and Scalability: In M2C communication mode, since the cloud servers have to maintain information for a huge number of smart objects in their databases, it is essential to perform authentication as fast as possible and in a period of time not related to the number of smart objects in the system. In an M2M communication mode, it is also necessary to have an authentication time that does not depend of the number of smart objects in the system.
  - (FR1b) Minimize Exchanged Messages: To ensure efficiency of communications, the number of messages and the quantity of data exchanged between parties during authentication must be minimized.
- (FR2) Lightweight: Due to the resource limitations of smart objects, an authentication protocol must be lightweight.
  - (FR2a) Computation: Cryptographic primitives used on smart objects must have a lower computational cost than traditional cryptographic algorithms (e.g., RSA and AES).
  - (FR2b) Storage: Data storage on smart objects must be minimized.
- (FR3) Resilience: To protect the system against Denial-of-Service (DoS) and desynchronization attacks, authentication protocols must be stateless-designed: i.e. communicating parties do not need to retain session information to ensure correct execution of the run.

An additional functional requirement for M2M communication modes is:

- (FR4) A distributed scheme: smart objects should be able to communicate directly and should not require the support of an on-line central server to execute the authentication phase.

3 Related work

In the literature, there are several authentication protocols for smart objects that can be classified according to M2C and M2M communication modes. They differ according to the type of communicating objects, the cryptographic methods used, the communication distances, and the application scope. Some authentication protocols for M2C and M2M communication models are discussed below.

3.1 Authentication in M2C communication mode

In M2C communication mode, the smart object can use proxies; entities with more resources that can carry out costly operations on their behalf, or can exchange directly with the cloud.

In [9], a proxy-based authentication and session key establishment in the context of IoT-enabled Ambient Assisted Living (AAL) is proposed. The authors use the Diffie-Hellman (DH) key exchange algorithm to establish a mutual authentication between a low-resource patient device in a smart home and a remote server. For communication, a (K, N) threshold scheme is used: the sender splits messages into N different pieces given to different proxies according the Lagrange formula [10] to enable receivers receiving at least K pieces to compute the original message (the scheme is resistant to $K-1$ colluding proxies but if K proxies collude the attacker can retrieve the original information). In [11], a Proxy Re-Encryption (PRE) technique is used to provide an authenticated key exchange to secure each communication between a smart object and an external powerful entity (e.g., a server in the cloud) or between two smart objects, without pre-sharing of a secret key between entities. In a basic way, in PRE, the proxy translates data encrypted by a key (initiator key) to data encrypted by another key (the receiver's key) without being able to access the unencrypted data. However, Nuñez et al. [12] have demonstrated that this protocol is vulnerable to break forward security, leading to key compromise impersonation and length-extension attacks. While such proxy-based proposals are interesting, because our M2C authentication protocols are designed for a direct connection, it is more relevant to present similar work. Contrary to proxy-based proposals, since all computations are done by resource-constrained smart objects, proposals often rely on lightweight cryptography algorithms such as ECC. In [13], Jin et al. present a mutual authentication protocol to improve safety in hospital drug delivery systems for patients. This protocol provides a mutual authentication between RFID tags (RFID bracelets worn by patients, RFID-labeled drug packaging) and the server (i.e. the cloud), while assuming that the communication channel between the tag and the RFID reader is not secure (whereas the one between reader and server is secure). To secure future exchanges over the wireless RFID communication channel, a session key is built during the authentication process between the two entities. This protocol is described in detail in Section 4.3.1 since one of our contributions is to propose two improvements. In recent years, several authors have proposed mutual authentication protocols between RFID tags and servers to establish a session key in order to efficiently secure communications: e.g. Zhao et al. in [14] for healthcare environments, Alamr et al. in [15], Liao et al. in [16] and Dinarvand et al. in [17] for any specific environment. During the setup phase the server generates a pair of ECC public-private keys for each party and/or a shared secret that will be used for mutual authentication during the authentication phase.

In addition to authentication protocols, it is worth mentioning another trend to secure data transmission from restricted devices to servers in the cloud, using Attribute Based Encryption (ABE) mechanisms. In [18], Sharma et al. propose a security platform for IoT infrastructure centralized data on the network layer. The purpose of this approach is to efficiently use the data available on the central server. The general idea is to collect data from several resources (such as medical devices and computers), encrypt these data and then add them to the central server. Access to the server to display data or perform a function on the data is achieved according to a set of attributes. To present their approach they use a real IoT application; an e-health system that collects data and records them in a centralized system. They combine both approaches, using Ciphertext-Policy Attribute Based Encryption (CP-ABE) to define the rules to access data and Functional Encryption (FE) to perform various functions on encrypted data. In [19], Li et al. propose a Lightweight Data Sharing Scheme (LDSS) for mobile cloud computing, to ensure secure data access while respecting the constraints imposed by mobile devices in terms of cryptography. They have adapted the CP-ABE technique of classic cloud computing to the mobile cloud environment by suitably modifying the structure of the access control tree and by outsourcing a large portion of the computational intensive access control tree transformation of CP-ABE from mobile devices to external proxy servers.

3.2 Authentication in M2M communication mode

As for the M2C communication mode, proxy-based authentication has been proposed. In [20], Porambage et al.

propose a scheme similar to [9] (presented above in M2C) to establish secure End-to-End (E2E) communication in the IoT between smart objects using the DH key exchange algorithm. In [21], another proxy-based authentication protocol for healthcare has been proposed by Amin et al. to preserve anonymity during a mutual authentication protocol in a Wireless Medical Sensor Network (WMSN). In the proposed architecture, authentication takes place between the medical staff's mobile devices and medical sensors through the proxy. However, in [22], Jiang et al. revised this protocol, which was vulnerable to stolen mobile device attack, desynchronization attack and sensor key exposure, to propose an improved E2E authentication protocol still using a proxy. Although such proxy-based proposals are interesting, because our M2M authentication protocol is designed for a direct connection, it is more relevant to present more closely related work.

However, after a careful analysis of the literature and examination of recent surveys on security for M2M [23], authentication protocols for IoT [24], and one on authentication protocols for the IoT but with a dedicated section for M2M [25], no M2M authentication protocol with a direct connection was identified.

4 Our proposals of M2C and M2M authentication protocols

This section contains the core contributions of this paper. To illustrate our two M2C authentication protocols and the M2M authentication protocol proposed, all notations used are introduced. Then, the attacker model and the automated verification tools used to validate the proposals are described. The first M2C authentication proposal is an improved version of that developed by Jin et al. [13] cited in Section 3, but for consistency it is presented with the same notation as that used for the other protocols. For each of our proposed protocols, an informal analysis is provided and a validation is conducted using two formal verification tools. All our proposed authentication protocols use a direct connection (i.e. without proxy) and are three-stage mutual authentication protocols comprising two phases:

- a setup phase: this first phase can be considered as off-line and is normally done once in the life of smart objects (e.g. at the initialization or personalization step). It mainly consists of a credentials provisioning step for all presented protocols.
- an authentication phase: this phase usually consists of three exchanges to achieve a mutual authentication of legitimate parties and is done each time authentication is required.

4.1 Protocol notations

Notations used to describe the different authentication protocols presented in the paper are summarized in Table 1.

4.2 Attacker model and formal verification tools

The considered attacker model and the formal verification tools used to validate the proposals are presented in the following sections.

4.2.1 Attacker model

The attacker model considered for our protocol is the widely used Dolev-Yao model [26]. Thus, the attacker can overhear, intercept and synthesize any message in a network. However, it cannot break the secure cryptographic primitives used in the construction of the protocol.

4.2.2 Formal verification tools

To formally validate the authentication protocols proposed, the AVISPA [27] and ProVerif [28] verification tools were used. Both use the Dolev-Yao attacker model.

AVISPA AVISPA, which stands for Automated Validation of Internet Protocols and Applications, provides a modular and expressive formal language for specifying protocols and their security properties. It integrates different back-ends that implement a variety of state-of-the-art automatic analysis techniques.

With AVISPA, the High-Level Protocol Specification Language (HLPSL) allows the writing of security protocol specifications using different roles. Some roles, called basic roles, serve to describe the actions of a single agent in a run of a protocol or sub-protocol. Others, called composed roles, instantiate these basic roles to model an entire protocol run, a session of the protocol between multiple agents, or the protocol model itself.

ProVerif ProVerif is an automatic cryptographic protocol verifier based on the Horn theory approach for intruders and protocol representation. Cryptographic protocols and associated security objectives are encoded in a formal manner, which is a variant of the applied pi calculus. This extension of the pi calculus with cryptography and support of types allows ProVerif to automatically verify user

Table 1 Notations used in protocols description

Notation	Signification
n	Number of smart objects in the system
S	Denotes a server
SO	Denotes generically smart object
T_i	Denotes a particular smart object T_i where $i \in [1, n]$
q, p	Two large prime numbers
$\mathbb{Z}_p^*$	A finite field
E	An elliptic curve defined over a finite field $\mathbb{F}_q$ by the equation $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_q$
P	A generator of E with order p
(q, a, b, p, P)	Denote the domain parameters defining E
N	Denote the size in bits of a private key using curve E ; $N = \log_2(p)$
H	Denotes a secure and collision resistant hash function
H_j	Denotes a secure and collision resistant hash function named H_j (used only in Jin et al.'s protocol)
$r \overset{\$}{\leftarrow} \mathbb{Z}_p^*$	Denotes that r is a random number $\in \mathbb{Z}_p^*$
$A + B$	According to the context, denotes the point addition between two points A and B or two scalars A and B
$A \cdot B$	According to the context, denotes the point multiplication between a scalar A and a point B or the scalar multiplication between two scalars A and B
$A \oplus B$	Denotes the XOR operation between values A and B
$A \| \| B$	Denotes the concatenation operation of values A and B
$A \stackrel{?}{=} B$	Denotes the verification of equality between A and B
$A = B$	Denotes variable A receives value of expression B
Id_{T_i}	Identity of T_i
HId_{T_i}	Key of hash entry related to the identity of T_i in database
x_S, P_S	Denote respectively private and public keys of Server S, where $P_S = x_S \cdot P$
x_i, P_i	Denote respectively private and public keys of T_i, where $P_i = x_i \cdot P$
$\sigma_i = (\bar{P}_i, z_i)$	Denotes the Schnorr signature of the public key, P_i, of T_i by the server: the server picks a random value $s_i \in \mathbb{Z}_p^*$ and computes $\bar{P}_i = s_i \cdot P$ and $z_i = s_i + H(P_i, \bar{P}_i) \cdot x_S$. The resulting signature is σ_i

cryptographic protocols specified either as rewrite rules or as equations theory.

4.3 M2C authentication protocols

In this section, two direct M2C authentication protocols are proposed and validated. The first improves Jin et al.'s ECC-based authentication protocol [13]. The second, which is mainly based on the use of hash functions, is introduced because it reduces both the storage and computational costs on smart objects as well as the computational costs on the server. In addition, this protocol is more scalable and flexible than the first one since it enables mutual authentication of smart objects and multiple servers as long as the latter know the identities of the smart objects and related keys for their hash entry in the database.

4.3.1 Improvements of Jin et al.'s authentication protocol

Before explaining our improvements to Jin et al.'s protocol, the original version is presented.

Jin et al.'s original authentication protocol As previously mentioned, Jin et al.'s authentication protocol [13] is ECC-based since it was designed for resource-constrained devices such as RFID tags. This protocol can be viewed as an M2C protocol, as the server can be hosted in the cloud and RFID tags can be viewed as smart objects.

Setup phase First, the server S generates and shares with other entities the system parameters (q, a, b, p, P) to use for an elliptic curve, E. The server then picks a random value $x_S \in \mathbb{Z}_p^*$ as its private key and computes its public key

as $P_S = x_S \cdot P$. Then, each smart object T_i receives from the server a unique identifier (named X_T in original paper and named here Id_{T_i}), which is a point on E, and the server's public key P_S.

Authentication phase This phase is composed of the three exchanges depicted in Fig. 2.

At the end of the authentication phase, the server S and the smart object T_i share a session key e that can be used to ensure the confidentiality, integrity and privacy of subsequent exchanges.

Security analysis The protocol being secure, we do not provide a security analysis, but interested readers can refer to that provided below for our improved version, since it is closely related. It is worth noting that in Jin et al.'s original protocol, the server S somehow implicitly authenticates the smart object T_i based on the existence of Id_{T_i} in the database. It is not a huge problem, since an attacker should not be able to compute the shared key e used for subsequent exchanges, but this issue does not exist in our proposal. Also, Jin et al. do not use the property that Id_{T_i} is a point on E. It may have been any string of bits with the same size. In fact, it is even more of a constraint for them since they need two hash functions, one whose output data size is that of the point on the curve (since to compute $Auth_i$ its output is used to mask a point) and one whose size of output is that of a scalar (since to compute $Auth_S$ its output is used in modular operation)

Two main improvements: reducing search complexity on the server side and computation on the smart object side In the original protocol presented in the previous section, since the server S stores all identifiers of smart objects in a database, to implicitly authenticate a smart object T_i, after having received $Auth_i$ and computed the smart object identifier, Id_{T_i}, the server has to check whether the smart object T_i exists in its database (Search Id_{T_i} step in Fig. 2). In terms of complexity that means that for n smart objects in the system, there may be at most n accesses to the server database if the identifier is stored at the last position. To reduce this linear complexity from $O(n)$ to a constant complexity, i.e. $O(1)$, inspired by Bonnefoi et al.'s protocol [29], our first improvement was to modify the protocol to add to the memory of smart object T_i at its enrollment in the setup phase, the value HId_{T_i} , which is the key of the hash entry in the database to access the related Id_{T_i}. In terms of memory consumption on the smart object, this is the same since the individual sizes of Id_{T_i} and HId_{T_i} are chosen to be half of the size of a point of E. With the addition of these two values our protocol runs faster on the server side and it has an explicit authentication (see security analysis below). It is worth noting that the message 2 (R_i, $Auth_i$, M) size is the same as that of message 2 in Jin et al.'s protocol.

A second improvement was to lighten the complex computations that smart objects have to achieve by modifying the message $Auth_S$ sent by the server: S now does the scalar multiplication that the smart object T_i had to do to authenticate the server. Thus one scalar multiplication is saved on the smart object but the message size is increased compared to that in the protocol designed by Jin et al.

A third minor improvement was to change the two hash functions used in the original version to only one, since having both was useless for security and they were consuming space in memory because of the code required to implement them, or Integrated Chip Circuit (ICC) space, if they were hardware-implemented.

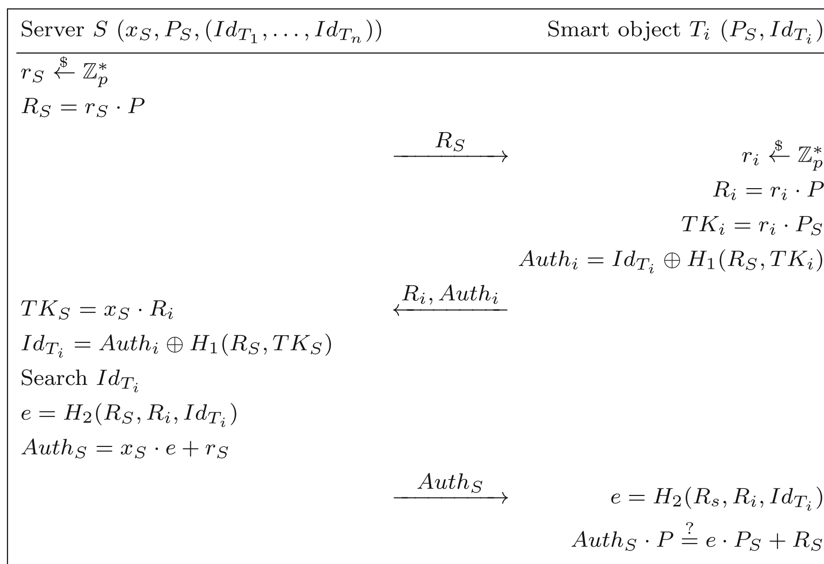


Fig. 2 Authentication phase for Jin et al.'s M2C protocol

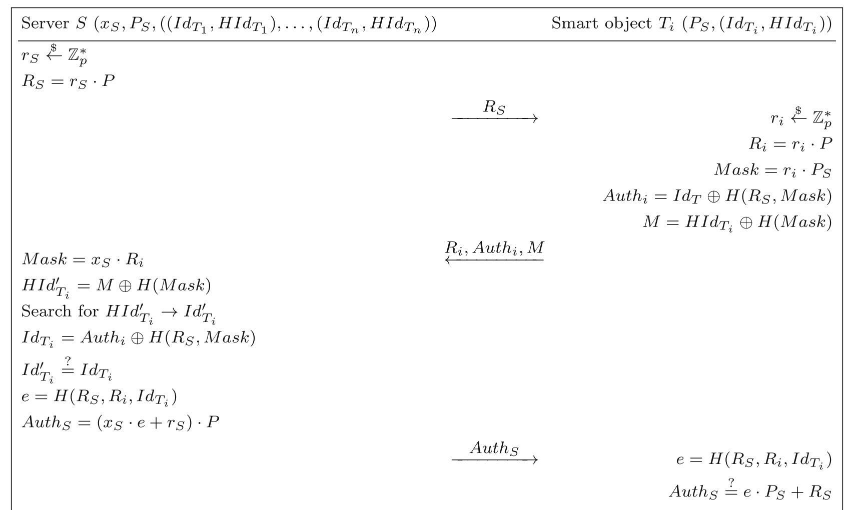


Fig. 3 Our improvements of the M2C Jin et al.'s authentication protocol

The improved version of the authentication protocol is depicted in Fig. 3.

Setup phase Similar to the setup phase in the original protocol, the server generates and shares with other entities the system parameters (q, a, b, p, P) to use for the elliptic curve, E. It then picks a random value $x_S \in \mathbb{Z}_p^*$ as its private key and computes its public key as $P_S = x_S \cdot P$. Then, for each smart object T_i, S generates a random Id_{T_i} as a smart object identifier and sends it along with $HId_{T_i} = H(Id_{T_i})$, the key of the database (hash table used to store smart objects' identifiers) and the server's public key P_S. It is worth noting that if there is a memory space constraint on T_i, HId_{T_i} does not need to be stored and can be computed by T_i itself at each authentication request, but it slows the authentication process.

Authentication phase In the following steps, we describe the interaction between the server S and a specific smart object T_i:

1. The server S picks a random value $r_S \in \mathbb{Z}_p^*$, computes $R_S = r_S \cdot P$ and sends R_S to T_i.
2. Upon reception, the smart object T_i first generates a random value $r_i \in \mathbb{Z}_n^*$, computes $R_i = r_i \cdot P$, $Mask = r_i \cdot P_S$ that is used to hide Id_{T_i} and HId_{T_i} in $Auth_i = Id_{T_i} \oplus H(R_S, Mask)$ and $M = HId_{T_i} \oplus H(Mask)$.
   Then it sends (R_i, $Auth_i$, M) to the server.
3. The server S computes $Mask = x_S \cdot R_i$, which is used to find $HId'_{T_i} = M \oplus H(Mask)$. Then, it looks for Id'_{T_i} in its database thanks to HId'_{T_i}.
   If Id'_{T_i} is not found, i.e. the entry does not exist, the server does not stop the session immediately to avoid leaking information to an attacker and it thus stops the session after a delay (i.e. in other words it should be programmed to answer within a fixed time for the three operations that are looking for Id'_{T_i}, computing Id_{T_i} and comparing it with Id'_{T_i}). Otherwise, it computes $Id_{T_i} = Auth_i \oplus H(R_S, Mask)$ and then compares Id_{T_i} with Id'_{T_i}. If the values are not equal, the server stops the session. Otherwise, the device is authenticated.
   The server then computes $e = H(R_S, R_i, Id_{T_i})$ and $Auth_S = (x_S \cdot e + r_S) \cdot P$. It sends $Auth_S$ to the smart object T_i.
4. The smart object T_i computes $e = H(R_S, R_i, Id_{T_i})$ and compares the received $Auth_S$ with $e \cdot P_S + R_S$. If the values are not equal, it stops the session. Otherwise the server is authenticated by the smart object and mutual authentication is achieved.

As in the original version, at the end of the authentication phase, the server S and the smart object T_i share a session key, e, that can be used to ensure the confidentiality, integrity and privacy of subsequent exchanges.

Before providing the security analysis of this protocol, it is worth mentioning that some optimization can be done on the smart object side between messages 2 and 3 (i.e. when the server S is verifying it is a legitimate smart object) since T_i can precompute $e = H(R_S, R_i, Id_{T_i})$ and $e \cdot P_S + R_S$, which accelerates the authentication of S on reception of message 3, i.e. $Auth_S$, since there is only a comparison to achieve. This trick can also be used with Jin et al.'s protocol but it was not mentioned (only very recently, in the RFID area, the primary scope of Jin et al., have protocol designers used the fact that the RFID tag is powered when it is in the field of the reader to perform precomputations [30]).

Security analysis In the following section, the protocol is analysed with regard to security requirements and some additional properties.

- Mutual authentication:
  - Smart object authentication: The adversary cannot generate, from the smart object T_i, a legitimate message $(R_i, Auth_i, M)$, since it is not able to obtain the coupled smart object identifier, Id_{T_i}, the related key, HId_{T_i}, and the random value r_i, which are used to compute both $Auth_i$ and M. However, the server S can on its side compute $Mask$ (since $x_S \cdot R_i$ is equal to $r_i \cdot P_S$) to obtain HId'_{T_i} and check whether the entry exists. It is worth noting that the server S must not yet stop the session if the entry does not exist; it must respond within a fixed time in order to not leak information to an attacker. To maintain the same level of security as in Jin et al.'s protocol, since we decided not to increase data storage on the smart object (sizes for our Id_{T_i} and HId_{T_i} were chosen to be the same as that for Id_{T_i} in Jin et al.), if the entry does not exist, S must wait as if it had achieved the matching of Id'_{T_i} and Id_{T_i}. If the entry exists, S receives the Id'_{T_i} and compares it with Id_{T_i}. If it does not match, the S stops the session. Otherwise, the smart object, T_i, is authenticated.
  - Server authentication: The adversary cannot generate a legitimate message from S, $Auth_S$, since it is not able to obtain the e that results from using a smart object identifier Id_{T_i}. Indeed a passive adversary can obtain R_S, and P_S is known; thus the strength of server authentication relies on e. The smart object T_i has to compute e to then check whether the $Auth_S$ received matches $e \cdot P_S + R_S$; if it does, the smart object authenticates S. It is worth noting that moving the scalar multiplication from $(e \cdot P_S + R_S) \cdot P$ from the smart object to the server (i.e. in $Auth_S$) to strengthen our authentication only relies on the ability to compute e, instead of that plus knowledge of x_S as in Jin et al.'s protocol. This is a choice and implementers of our protocol can decide not to use this improvement, although the strength is similar.
- Confidentiality, integrity and privacy: At the end of the protocol, both parties, S and T_i, share a fresh session key, e, to which both have contributed, since it is based on random values r_S, r_i for each session and Id_{T_i}. S and T_i can use it to derive keys to ensure the confidentiality, integrity and privacy of subsequent exchanges.
- Availability:
  - Desynchronization attack resistance: In the proposed protocol, since there is no key update step and the protocol has a stateless design, keys cannot be desynchronized and the protocol can be resumed from scratch at any step of the authentication.
  - DoS attack resistance: The proposed protocol being stateless, each transaction is considered as a new one. On the smart object side, a new authentication transaction does not consume more memory space than the previous one. It uses computational resources but this is unavoidable. The worst case on the smart object side depends on the implementation and type of memory used: if the smart object uses non-volatile memory technology like Flash or EEPROM to store its intermediate computations, the memory cells may be dead after 10^5 writing cycles; however, if it uses FRAM it will last 10^{12} cycles and if it is RAM only the number of cycles is infinite. On the server side, since the server must be able to handle authentications from several smart objects at the same time, each new authentication transaction consumes some memory resources, but in the case of a DoS attack, our efficient searching mechanism limits much of the impact since the server does not spend its time making computations and querying the database for fake HId'_{T_i}.
- Scalability: Thanks to our more efficient searching mechanism, the server S computes the $Mask$, then checks the existence of HId'_{T_i} and potentially verifies the matching of Id'_{T_i} and Id_{T_i} in a constant time, i.e. $O(1)$.
- Smart object's anonymity: In the proposed protocol, the coupled smart object identifier, Id_{T_i}, and the related key, HId_{T_i} are hidden with a function using $Mask$. While an adversary can obtain R_S and P_S is known, it cannot compute $Mask$ without the random value r_i and thus it cannot compute Id_{T_i} or HId_{T_i}. In addition, in each new session, the server S and the smart object T_i generate separate new random values, r_S and r_i. Thus the adversary cannot trace the smart object's location and our protocol provides anonymity.

- Formal analysis with AVISPA
  In the following paragraphs, important points of the protocol specification and the results are highlighted.
  - Protocol specification
    The HLPSL script of our proposed protocol is shown in Appendix A.1.1. There are two basic roles, S and T, which explain the activity of $Server$ and T_i. The fundamental concept of the protocol is to keep the values Id_{T_i}, HId_{T_i}, x_S, r_S and r_i `secret` through the protocol authentication phase between S and T. Mutual authentication is achieved via `witness` and `request` goals. Details are provided in the Appendix.
  - Verification results
    The results after running our protocol coded in HLPSL are given in Table 2.

In Table 2, AVISPA outputs SAFE from two of its four back-ends On-the-Fly Model-Checker (OFMC) and CL-based Model-Checker (CL-AtSe), while SAT-based Model-Checker (SATMC) and Tree Automata, based on Automatic Approximations for the Analysis of Security Protocols (TA4SP) give INCONCLUSIVE results due to unsupported operations, which means that AVISPA cannot find any attack on our protocol.

- Formal analysis with ProVerif
  In the following paragraphs, important points of the protocol specification and the results are highlighted.
  - Protocol specification
    The applied pi calculus scripts are shown in Appendix A.2.1. As for AVISPA, there are two basic roles `SERVERS` and `SOI`, which explain the activity of $Server$ and T_i. Secret values are checked using queries. Mutual authentication between T_i and $Server$ is modeled using events that are mapped in the `SOI` and `SERVERS` subprocesses and queries. Details are provided in the Appendix.
  - Verification results
    The results available in Appendix A.2.2 show that the secrecy of `idt`, `hidt`, `xs`, `ri`, `rs` are preserved by the protocol and mutual authentication between `SERVERS` and `SOI` is achieved.

Table 2 AVISPA validation results

AVISPA Engine	Result
$OFMC$	SAFE
$CL-AtSe$	SAFE
$SATMC$	INCONCLUSIVE
$TA4SP$	INCONCLUSIVE

4.3.2 Hash-based authentication protocol

In this section, we present another authentication protocol, also using ECC, but which is more hash-based. The proposed protocol consists of two phases, i.e., the setup phase and the authentication phase. The main advantage of this is to be more scalable and flexible than the previous protocol since it enables mutual authentication of smart objects and multiple servers as long as the latter know the identities of the smart objects. It means that if a server crashes, authentication can be done by another server if it knows the smart object identity and its related key of hash entry.

The authentication phase of the proposed protocol is presented in Fig. 4.

Setup phase In this phase, the server generates and shares with other entities the system parameters (q, a, b, p, P) to use for the elliptic curve, E. At this stage, for each smart object T_i, S generates a random Id_{T_i} as smart object identifier and sends along with it $HId_{T_i} = H(Id_{T_i})$, the key of the database (hash table used to store smart objects' identifiers). If there is memory space constraint on T_i, HId_{T_i} does not need to be stored and can be computed by T_i itself at each authentication request, but this slows the authentication process.

Authentication phase

1. The server S picks a random value $r_S \in \mathbb{Z}_p^*$, computes $R_S = r_S \cdot P$ and sends R_S to T_i.
2. Upon reception, the smart object T_i first generates a random value $r_i \in \mathbb{Z}_n^*$, computes $R_i = r_i \cdot P$ and $K = r_i \cdot R_S$, which is used to hide HId_{T_i} in $M = HId_{T_i} \oplus K$ (where in fact only the x coordinate of point K is used to mask – we might have also used $H(K)$ instead but it would require an additional call to the hash function). It also computes $Auth_i = H(Id_{T_i} \| 2 \| K)$ and then sends it to $(R_i, Auth_i, M)$ to the server.
3. The server, S, first computes $K = r_S \cdot R_i$, which is used to find $HId'_{T_i} = M \oplus K$. Then, it looks for Id'_{T_i} in its database using HId'_{T_i}.
   If Id'_{T_i} is not found, i.e. the entry does not exist, the server S does not stop the session immediately to avoid leaking information to an attacker; it thus stops the session after a delay (i.e. in other words it should be programmed to answer within a fixed time for the three operations that are looking for Id'_{T_i}, computing $Auth'_i$ and comparing it with $Auth_i$). Otherwise, it computes $Auth'_i = H(Id'_{T_i} \| 2 \| K)$ and then compares

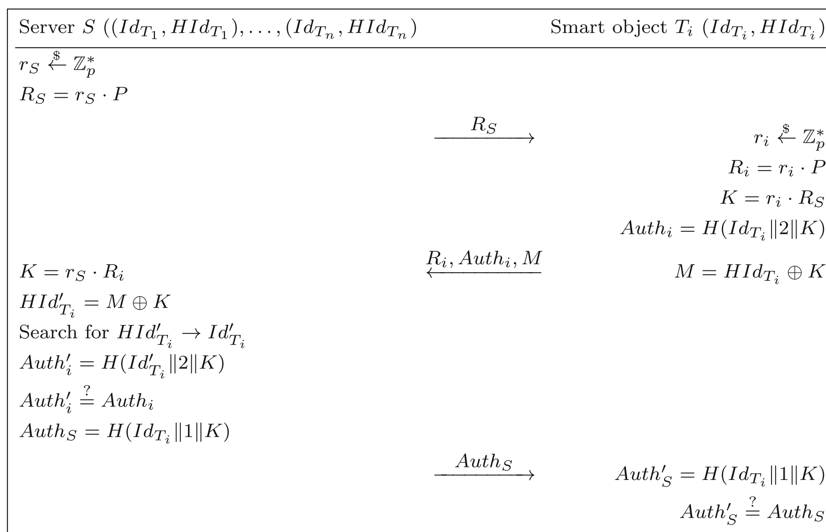


Fig. 4 Hash-based M2C authentication protocol

$Auth_i$ with $Auth'_i$. If the values are not equal, the server stops the session. Otherwise, the smart object is authenticated.

The server S computes $Auth_S = H(Id_{T_i} \| 1 \| K)$ and sends it to the smart object T_i.

4. Upon reception, the smart object T_i computes $Auth'_S = H(Id_{T_i} \| 1 \| K)$ and compares $Auth_S$ and $Auth'_S$. If the values are not equal, it stops the session. Otherwise, the server S is authenticated and mutual authentication is achieved.

At the end of the authentication phase, the server S and the smart object T_i share a session key, K, which can be used to ensure the confidentiality, integrity and privacy of subsequent exchanges.

Security analysis In the following, the protocol is analysed with regard to the security requirements and some additional properties.

- Mutual authentication:
  - Smart object authentication: The adversary cannot generate a legitimate message from a smart object T_i ($R_i, Auth_i, M$) since it is not able to obtain the coupled smart object identifier, Id_{T_i}, related key, HId_{T_i}, and the random value r_i, which are used to compute both $Auth_i$ and M. However the server S can on its side compute K (since $r_S \cdot R_i$ is equal to $r_i \cdot R_S$) to obtain HId'_{T_i} and check whether the entry exists. The server S must not yet stop the session if the entry does not exist; it must respond within a fixed time in order to not leak information to attacker. If the entry exists, S obtains Id'_{T_i} and compares it with Id_{T_i}. If it does not match, the S stops the session. Otherwise, the smart object, T_i is authenticated.
  - Server authentication: The adversary cannot generate a legitimate message from S $Auth_S$ since it is not able to obtain the smart object identifier Id_{T_i} and K. To authenticate the server S, the smart object T_i has to compute $Auth'_S$ to then check whether the $Auth_S$ received matches or not.
- Confidentiality, integrity and privacy: At the end of the protocol, both parties S and T_i share a fresh session key, K, to which both have contributed, since it is based on random values r_S, r_i for each session. S and T_i will be able to use it to derive keys to ensure the confidentiality, integrity and privacy of subsequent exchanges.
- Availability:
  - Desynchronization attack resistance: In the proposed protocol since there is no key update step and the protocol has a stateless design, keys cannot be desynchronized and the protocol can be resumed from scratch at any step of the authentication.
  - DoS attack resistance: The proposed protocol being stateless, each transaction is considered as a new one. On smart object side, a new authentication transaction does not consume more memory space than the previous one. It uses computational resources but this is unavoidable. The worst case on the smart object side depends on the implementation and the type of memory used: if the smart object uses non-volatile memory technology

like Flash or EEPROM to store its intermediate computations, the memory cells may be dead after 10^5 writing cycles; however if it uses FRAM it will last 10^{12} cycles and if it is RAM only the number of cycles is infinite. On server side, since the server must be able to handle authentication from several smart objects at the same time, each new authentication transaction consumes some memory resources, but in the case of a DoS attack, our efficient searching mechanism limits much of the impact since the server does not spend its time making computations and querying the database for fake HId'_{T_i}.

- Scalability: Thank to the introduced efficient searching mechanism, the server S computes the K, then checks the existence of HId'_{T_i}, and potentially verifies the matching of Id'_{T_i} and Id_{T_i} within a fixed time, i.e. $O(1)$. In addition, this protocol enables mutual authentication of smart objects and multiple servers as long as the latter know the identities of the smart objects. It means that if a server crashes, authentication can be done by another server if it knows the smart object's identity and its related key of hash entry.
- Smart object's anonymity: In the proposed protocol, while the coupled smart object identifier, Id_{T_i} and the related key, HId_{T_i} are hidden with a function using K and a hash function, an active adversary can compute the K (using the r_S generated and the R_i received). Thus he can compute HId'_{T_i}. However, he cannot compute Id_{T_i}. Thus the adversary can trace the smart object's location but cannot obtain its identity.

- Formal analysis with AVISPA
  In the following paragraphs, important points of the protocol specification and the results are highlighted.
  - Protocol specification
    The HLPSL script of our proposed protocol is shown in Appendix A.1.2. There are two basic roles, S and T, which explain the activity of *Server* and T_i. The fundamental concept of the protocol is to keep the values Id_{T_i}, HId_{T_i}, r_S and r_i `secret` through the protocol authentication phase between S and T. Mutual authentication is achieved via `witness` and `request` goals. Details are provided in the Appendix.
  - Verification results
    The results after running our protocol coded in HLPSL are given in Table 3.

Table 3 AVISPA validation results

AVISPA Engine	Result
$OFMC$	SAFE
$CL-AtSe$	SAFE
$SATMC$	INCONCLUSIVE
$TA4SP$	INCONCLUSIVE

In Table 3, AVISPA outputs SAFE from two of its four back-ends On-the-Fly Model-Checker (OFMC) and CL-based Model-Checker (CL-AtSe), while SAT-based Model-Checker (SATMC) and Tree Automata, based on Automatic Approximations for the Analysis of Security Protocols (TA4SP) give an INCONCLUSIVE result due to unsupported operations, which means that AVISPA cannot find any attack on our protocol.

- Formal analysis with ProVerif
  In the following paragraphs, important points of the protocol specification and the results are highlighted.
  - Protocol specification
    The applied pi calculus scripts are shown in Appendix A.2.3. As for AVISPA, there are two basic roles `SERVERS` and `SOI`, which explain the activity of *Server* and T_i. Secret values are checked using queries. Mutual authentication between T_i and *Server* is modeled using events that are mapped in the `SOI` and `SERVERS` subprocesses and queries. Details are provided in the Appendix.
  - Verification results
    The results available in Appendix A.2.4 show that the secrecy of `idt`, `hidt`, `ri`, `rs` are preserved by the protocol and mutual authentication between `SERVERS` and `SOI` is achieved.

4.4 M2M authentication protocol: our ECC-based authentication protocol

In this section, our new M2M authentication protocol based on ECC is proposed and validated. Like the previous protocols, it comprises two phases; a setup phase and an authentication phase.

Setup phase In this phase, the certification authority server generates and shares with other entities the system parameters (q, a, b, p, P), to use for the elliptic curve, E. Then, each smart object T_i computes its private/public keys

(x_i, P_i) and the certification authority server S does the same for its private/public keys (x_S, P_S).

Next, each smart object T_i requests the server to certify its public key using a Schnorr signature [31]: i.e., the server picks a random value $s_i \in \mathbb{Z}_p^*$ and computes $\bar{P}_i = s_i \cdot P$ and $z_i = s_i + H(P_i, \bar{P}_i) \cdot x_S$. The resulting signature, which is $\sigma_i = (\bar{P}_i, z_i)$ is then provided to the smart object T_i.

Finally, at the end of this phase, the smart object T_i stores the system parameters, its keys pair (x_i, P_i), the signed public key σ_i and the server public key P_S in its memory.

Authentication phase The proposed M2M authentication protocol enables authentication between any smart objects that want to communicate. In the following, the steps to achieve two smart objects' mutual authentication, as depicted in Fig. 5, are described.

1. The first smart object T_i picks a random value $r_i \in \mathbb{Z}_p^*$, and computes $R_i = r_i \cdot P_i$. Then, T_i sends (R_i, P_i, σ_i) to the second smart object T_j.
2. Upon reception, T_j first checks the Schnorr signature to validate the received public key, P_i. If validation fails, T_j stops the session. Otherwise, T_j picks a random value $r_j \in \mathbb{Z}_p^*$ and computes $R_j = r_j \cdot P_j$ and $Auth_j = x_j \cdot R_j$. T_j then sends the message $(R_j, Auth_j, P_j, \sigma_j)$ to T_i.
3. T_i verifies the received Schnorr signature of T_j's public key. If the verification fails, T_i stops the session. Otherwise, T_i computes $Auth'_j = r_i \cdot x_i \cdot P_j$ and checks whether the received $Auth_j$ matches. If they are not equal, T_i stops the session; otherwise, T_j is authenticated to T_i. Finally, T_i computes $Auth_i = x_i \cdot R_j$ and sends it to T_j.
4. Upon reception, T_j computes $Auth'_i = r_j \cdot x_j \cdot P_i$ and checks whether the received $Auth_i$ matches.

   If they are not equal, T_j stops the session. Otherwise, T_i is authenticated to T_j and mutual authentication is achieved.

To obtain a shared session key, K, each smart object, T_i and T_j, needs to compute it as follows: $r_i \cdot R_j$ and $r_j \cdot R_i$ respectively. This process costs each smart object one more scalar multiplication.

Security analysis In the following, the protocol is analysed with regard to security requirements and some additional properties.

- Mutual authentication:
  - T_j authentication: The adversary cannot generate a legitimate message from a smart object T_j $(R_j, Auth_j, P_j, \sigma_j)$ since it cannot obtain x_j, the private key belonging to T_j, used in the computation of $Auth_j$. Although obtaining legitimate R_j, P_j and σ_j is easy, when T_i computes $Auth'_j$ and compares it to the received value $Auth_j$, it cannot match if the correct private key x_j has not been used, since $Auth'_j = r_i \cdot x_i \cdot P_j = r_i \cdot x_i \cdot (x_j \cdot P) = r_i \cdot x_j \cdot (x_i \cdot P) = r_i \cdot x_j \cdot P_i = x_j \cdot (r_i \cdot P_i) = x_j \cdot R_i = Auth_j$. Thus if $Auth_j$ and $Auth'_j$ match, the smart object T_j is authenticated.
  - T_i authentication: Similar explanations apply to T_i authentication.
- Confidentiality, integrity and privacy: At the end of the protocol, both parties, T_i and T_j, can share a fresh session key, K, by respectively computing $r_i \cdot R_j$ and

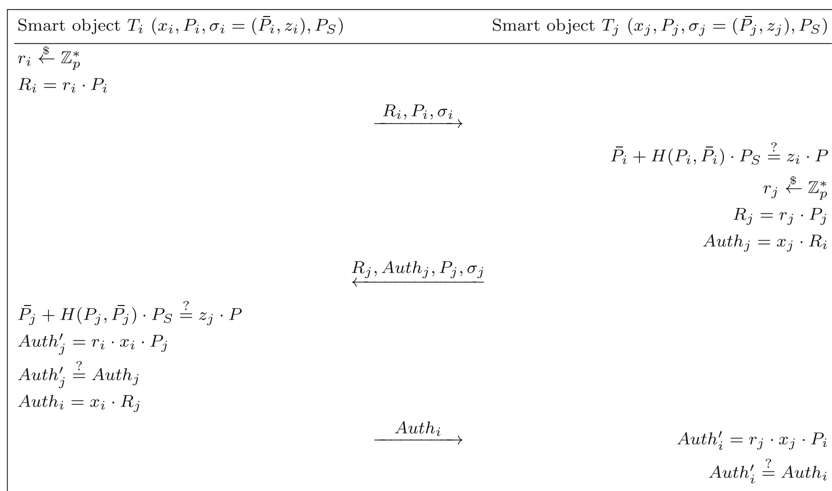


Fig. 5 ECC-based M2M authentication protocol

$r_j \cdot R_i$), which both have contributed to, since it is based on random values r_i, r_j for each session. T_i and T_j can use this to derive keys to ensure the confidentiality, integrity and privacy of subsequent exchanges.

- Availability:
  - Desynchronization attack resistance: In the proposed protocol, since there is no key update step and the protocol has a stateless design, keys cannot be desynchronized and the protocol can be resumed from scratch at any step of the authentication.
  - DoS attack resistance: The proposed protocol being stateless, each transaction is considered as a new one. For each party, a new authentication transaction does not consume more memory space than the previous one. It uses computational resources, but this is unavoidable. The worst case depends on the implementation and type of memory used: if the smart objects use non-volatile memory technology like Flash or EEPROM to store their intermediate computations, the memory cells may be dead after 10^5 writing cycles; however if they use FRAM it will last 10^{12} cycles and if it is RAM only the number of cycles is infinite.
- Scalability: As long as the certification authority server is enabled for the setup phase, it is possible to add new smart objects to the system. When they are in the system, since smart objects authenticate in a decentralized manner, the system is scalable.
- Smart objects' anonymity: In the proposed protocol, since P_i and σ_i are always the same, it is possible to trace a smart object and identify it (as long as we consider a public key as an identifier), but this is not one of our requirements. Despite this issue, the privacy of subsequent exchanges between smart objects is ensured using the shared key K.
- Formal analysis with AVISPA

  In the following paragraphs, important points of the protocol specification and the results are highlighted.

  - Protocol specification

    The HLPSL script of our proposed protocol is shown in Appendix A.1.3. There are two basic roles, A and B, which explain the activity of T_i and T_j nodes. The fundamental concept of the protocol is to keep the values x_i, x_j, r_i and r_j `secret` through the protocol authentication phase between T_i and T_j. Mutual authentication is achieved via `witness` and `request` goals. Details are provided in the Appendix.
  - Verification results

    The results after running our protocol coded in HLPSL are given in Table 4.

Table 4 AVISPA validation results

AVISPA Engine	Result
$OFMC$	SAFE
$CL-AtSe$	SAFE
$SATMC$	INCONCLUSIVE
$TA4SP$	INCONCLUSIVE

In Table 4, AVISPA outputs SAFE from two of its back-ends; On-the-Fly Model-Checker (OFMC) and CL-based Model-Checker (CL-AtSe), while the SAT-based Model-Checker (SATMC) and Tree Automata, based on Automatic Approximations for the Analysis of Security Protocols (TA4SP) give an INCONCLUSIVE result due to unsupported operations, which means that AVISPA cannot find any attack on our protocol.

- Formal analysis with ProVerif

  In the following paragraphs, important points of the protocol specification and the results are highlighted.

  - Protocol specification

    The applied pi calculus scripts are shown in Appendix A.2.5. As for AVISPA, there are two basic roles `SOA` and `SOB`, which explain the activity of T_i and T_j. Secret values are checked using queries. Mutual authentication between T_i and T_j is modeled using events that are mapped in the `SOA` and `SOB` subprocesses and queries. Details are provided in the Appendix.
  - Verification results

    The results available in Appendix A.2.6 show that the secrecy of `xa`, `xb`, `ra` and `rb` are preserved by the protocol and mutual authentication between `SOA` and `SOB` is achieved.

5 Protocols evaluation

Instead of arbitrarily choosing particular smart objects to evaluate the efficiency of our proposals, we decided to make a formal analysis based on the domain parameters of the curves used and the number of smart objects in the

Table 5 Comparison of smart object's computational cost

Protocol	Computational cost
Liao et al. [16]	$5 * T_{sm} + 3 * T_{add}$
Zhao [14]	$5 * T_{sm} + 3 * T_{add}$
Alamr et al. [15]	$4 * T_{sm} + 1 * T_{add}$
Dinarvand et al. [17]	$3 * T_{sm} + 2 * T_{add}$
Jin et al. [13]	$4 * T_{sm} + 1 * T_{add}$
Our improvements of Jin et al.	$3 * T_{sm} + 1 * T_{add}$
Our hash-based authentication proposal	$2 * T_{sm}$

system. However, to assess the validity of our proposals, we conducted some practical time measurements of the most expensive operation used in the protocols, scalar multiplication, for different curves on a highly resource-constrained device, the Multos IoT Trust Anchor, which runs a secure operating system on a secure hardware microcontroller. To confirm our results, we also conducted measurements on a second target using a similar model of microcontroller but running Java Card technology under a smart card form factor.

5.1 Formal performance analysis

In this section, we analyze the performance of the proposed protocols according to computational costs, communication costs, storage requirements and performance comparisons.

For this we consider a system with n smart objects. Depending on the domain parameters chosen for E, the size of the point on the curve and the scalar used change according the following formula: The size in bits of a point is $2N$ and the size of a scalar is N. The domain parameters (q, a, b, p, P) size in bits is usually $6N$ (bit sizes of q, a and b are often similar to that of p, i.e. N, and the generator P is a point, thus its bit size is $2N$). We assume that both Id_{T_i} and $H(Id_{T_i})$ are scalar, i.e. the bit size of each is N.

T_{sm} and T_{add} denote the time elapsed to achieve a scalar multiplication and points addition, respectively.

5.1.1 Analysis of M2C authentication protocols

In this section we compare our M2C proposals with selected protocols (from Section 3): Dinarvand et al.'s [17], Liao et al.'s [16], Zhao's [14], Alamr et al.'s [15], and Jin et al.'s [13].

Computation costs The comparisons between smart objects and server computational costs for the authentication phase for our proposals and selected other protocols are summarized in Tables 5 and 6 respectively.

From Table 5, the two best protocols in term of computational cost for smart objects are our proposals: 1) the hash-based authentication protocol; and 2) our improvements of Jin et al.'s protocol. Compared to Jin et al. we win with regard to scalar multiplication on smart objects and lose on the server side. From Table 6, the two best protocols in term of computational cost for the server are our proposals: 1) the hash-based authentication protocol; and 2) our improvements of Jin et al.'s protocol even if we have a scalar multiplication, thanks to our improvements relating to database search complexity.

In our two M2C proposals (i.e. improvements of Jin et al.'s protocol and the hash-based authentication protocol) we improved the transaction time on the database on the server side during smart object authentication by reducing the database computational search cost to $O(1)$ compared to $O(n)$ for the related work.

Communication costs To determine communication costs, we computed the total bit size of transmitted messages during the authentication phase and we also counted the number of flows.

Table 7 summarizes the communication costs of the different protocols.

From Table 7, the two best protocols in term of communication cost for the server are: 1) the hash-based authentication protocol; and 2) the original version of Jin et al.'s protocol, since in our improvements of the latter we decided to do the scalar multiplication on the server

Table 6 Comparison of server's computational cost

Protocol	Computational cost	Database search complexity
Liao et al. [16]	$5 * T_{sm} + 3 * T_{add}$	$O(n)$
Zhao et al. [14]	$5 * T_{sm} + 3 * T_{add}$	$O(n)$
Alamr et al. [15]	$5 * T_{sm} + 1 * T_{add}$	$O(n)$
Dinarvand et al. [17]	$3 * T_{sm} + 2 * T_{add}$	$O(n)$
Jin et al. [13]	$2 * T_{sm}$	$O(n)$
Our improvements of Jin et al.	$3 * T_{sm}$	$O(1)$
Our hash-based authentication proposal	$2 * T_{sm}$	$O(1)$

Table 7 Comparison of communication cost

Protocol	SO→S (bits)	S→SO (bits)	Total (bits)	# of flows
Liao et al. [16]	$4N$	$4N$	$8N$	3
Zhao [14]	$4N$	$4N$	$8N$	3
Alamr et al. [15]	$4N$	$6N$	$10N$	3
Dinarvand et al. [17]	$5N$	$4N$	$9N$	4
Jin et al. [13]	$4N$	$3N$	$7N$	3
Our improvements of Jin et al.	$4N$	$4N$	$8N$	3
Our hash-based authentication proposal	$4N$	$3N$	$7N$	3

instead of the smart object (i.e. a point is sent instead of a scalar). However it would be possible to transmit only the x coordinate of $Auth_S$ without jeopardizing security if we want to be considered the two best candidates for communication costs.

Storage requirements The storage requirement is defined as the space used to store data in system communicating entities. In any protocol, it means the domain parameters and the additional data (such as credentials, shared keys, and identities) stored at the end of the setup phase. Table 8 summarizes the storage requirements of the different protocols.

From Table 8, the three best protocols in terms of storage requirements for smart objects and for servers are: 1) our hash-based authentication protocol; and then 2) tied, our improvements of Jin et al.'s protocol and the original Jin et al. 's protocol. We did not consider Alamr et al.'s protocols due to the storage requirements on the smart object.

Summary According to the performance criteria analysed, the two most efficient protocols are the two M2C protocols we proposed. The hash-based authentication protocol outperforms all others on all criteria. Although our improvements of Jin et al.'s protocol mean there is one more computation on the server, this is negligible compared to the time saved by introducing our method to reduce the database search operation from linear complexity ($O(n)$) to a constant complexity access ($O(1)$). We also explained how the communication cost from server to smart object can be reduced to that of Jin et al.'s protocol without jeopardizing security.

To conclude, the hash-based authentication protocol is the best solution for an M2C communication mode, since it is more flexible and more scalable than the others we examined.

5.1.2 Analysis of M2M authentication protocol

As explained in Section 3 there is no similar (direct connection, i.e. without proxy) authentication protocol proposed in the literature and thus we could only analyse our own.

Unlike the M2C communication mode, where the smart object communicates only with the server, in M2M authentication protocols, a smart object usually needs to have basic information about other smart objects present in the system to allow it to communicate. Thus, in our ECC-based authentication protocol to reduce the storage requirements of smart objects and to make it possible to dynamically communicate with new smart objects, in addition to their own credentials, on each smart object we only store the certification authority server's public key.

Table 8 Comparison of storage requirements

Protocol	SO (bits)	S (bits)
Liao et al. [16]	$6N + 5N$	$6N + 3N + 3N \cdot n$
Zhao [14]	$6N + 5N$	$6N + 3N + 3N \cdot n$
Alamr et al. [15]	$6N + 5N$	$6N + 3N + 2N \cdot n$
Dinarvand et al. [17]	$6N + 5N$	$6N + 3N + 3N \cdot n$
Jin et al. [13]	$6N + 4N$	$6N + 3N + 2N \cdot n$
Our improvements of Jin et al.	$6N + 4N$	$6N + 3N + 2N \cdot n$
Our hash-based authentication proposal	$6N + 2N$	$6N + 2N \cdot n$

Table 9 ECC-based authentication protocol performance analysis per smart object

Computational cost	$5 * T_{sm} + 1 * T_{add}$
Communication cost (bits)	$9N$
Storage requirements (bits)	$6N + 8N$

This allows the smart object to verify the authenticity of the public keys of other smart objects without needing to store them in its memory.

The performance analysis provided in Table 9 shows that the proposed protocol is more than suitable for an M2M communication mode.

5.2 Practical analysis

To evaluate the practical performance of our proposals, time measurements of different operations were conducted on two targets: the Multos IoT Trust Anchor [32] and the Universal JCard [33]. The Multos IoT Trust Anchor being designed for IoT applications, most operations used in the proposed protocols were subjected to performance measurements: hash (SHA-1), xor, multiplication, modular multiplication and scalar multiplication. The second target, the Universal JCard, was only used to confirm that the most expensive operation, i.e. the scalar multiplication, could reach the same performance on the same hardware platform but running another operating system.

5.2.1 Overview of the targets

We chose to perform our experimental measurements on the two aforementioned platforms for several reasons. The two devices are both highly resource-constrained platforms composed of a secure ICC and a secure system. The Multos IoT Trust Anchor is the first initiative from the Multos Consortium [34] to port the Multos smart card Operating System (OS) on a secure embedded ICC to address the IoT domains to enable the running of secure applications. Since the hardware used for this platform was one model, SLE78CLUFX5000PHM [35], of the SLE78 series from Infineon, which is often used in smart card products, we chose to study the performance of the most expensive cryptographic operation, the scalar multiplication, on another model, SLE78CLFX4000P [36], of the SLE78, running a Java Card operating system implementation. This was also motivated by the recent choice of Oracle Inc. to target IoT devices with Java Card technology [37], as can be seen with release 3.1 of the Java Card specifications [38, 39]. Both SLE78 ICCs contain several security hardware sensors (temperature, light, voltage, frequency) and protections (I^2-shield, which is an Infineon-specific shielding technique combined with secure wiring of critical security signals to avoid manipulation or eavesdropping by an attacker). These 16-bit security controllers integrate the Integrity Guard [40] which is, in brief, a dual CPU approach constantly, checking each other to establish whether the other unit is functioning correctly, and the SOLID FLASH™ [41], which is a technology that replaces mask ROM with a Flash memory to provide the flexibility required to answer time-to-market needs.

Multos IoT Trust Anchor The Multos IoT Trust Anchor consists of a secure ICC, i.e. SLE78, which runs the Multos OS. This OS is known to be one of the most secure in the smart card industry: i.e. sensitive operations are conducted in a state-of-the-art manner (e.g. operations are done in "constant time" to avoid information leakage). The target is available to developers in two footprints [42]: a DIP32 format that can be plugged into a development board to provide access to the GPIO, I2C, SPI, serial and contact and contactless smart card interfaces; and a nano board that breaks out all the used pins for the same interfaces. Footprints respectively run the M5-P22 (aka Multos 4.5.3) [43] and M5-P19 (aka Multos 4.5.1) [44] families of Multos OS implementations. The development board for the DIP32 footprint provides different features to ease the development: an on-board 3.3V regulator (for USB or external DC power), power on/off switch, USB to serial interface, LEDs for Tx/Rx, a general purpose LED, a push button switch, and header pins to access the different interfaces. In the following, we used the DIP32 format plugged in to the development board. To develop applications, we used the Multos SmartDeck [45], which enabled us to easily compile our programs in C-language for the Multos targets.

Universal JCard The Universal JCard is available on a smart card form factor with ISO7816 contact and ISO14443 contactless interfaces. It consists of a secure ICC, i.e. SLE78, which runs the Java Card OS. Java Card [46] is a multiapplication technology for memory-constrained devices that enables secure running of applications from different providers [47]. The implementation available on the Universal JCard is Oracle's reference implementation of Java Card version 3.0.1. To develop applications, so called applets, we used Infineon's development software [48]. JCIDE stands for "Java Card IDE", which relies on the

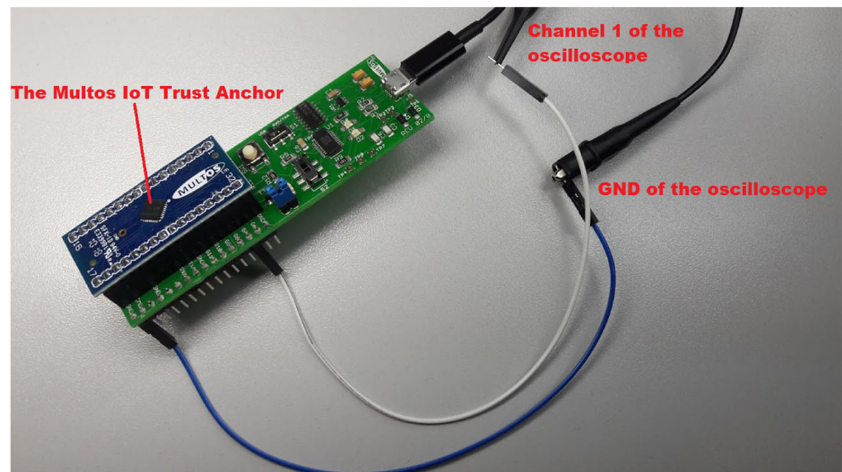


Fig. 6 Multos IoT Trust Anchor running our application and connected to channel 1 of our oscilloscope

Java Card Development Kit [49] to build the distribution file, i.e. the `cap` file. We also used it to load this file onto the card, using the GlobalPlatform-compliant integrated tool.

It is worth noting that interaction with a smart card is limited to APDU protocol on contact or contactless interfaces; i.e. commands sent by the reader to the card and responses received from the card.

5.2.2 Methodology to measure performance on the Multos IoT Trust Anchor

To measure the internal timing of Multos OS implementation of our device for a hash function, xor, multiplication, modular multiplication and scalar multiplication, we used API calls. In our implementation, it was not possible to directly access the scalar multiplication primitive and since only Short-Weierstrass curves were supported and there were constraints on the specification of domain parameters, we only did time measurements for several elliptic curves (secp192k1, secp192r1, Wei22519, secp256k1, secp256r1, secp384r1 and secp521r1).

In short, we implemented a very simple application calling on API functions, using the operations hash function, xor, multiplication and scalar multiplication: i.e. `multoSHA1`, `multosXor`, `multosMultiply` and `multosECDH` respectively. Since the modular multiplication was not directly accessible via API functions, we

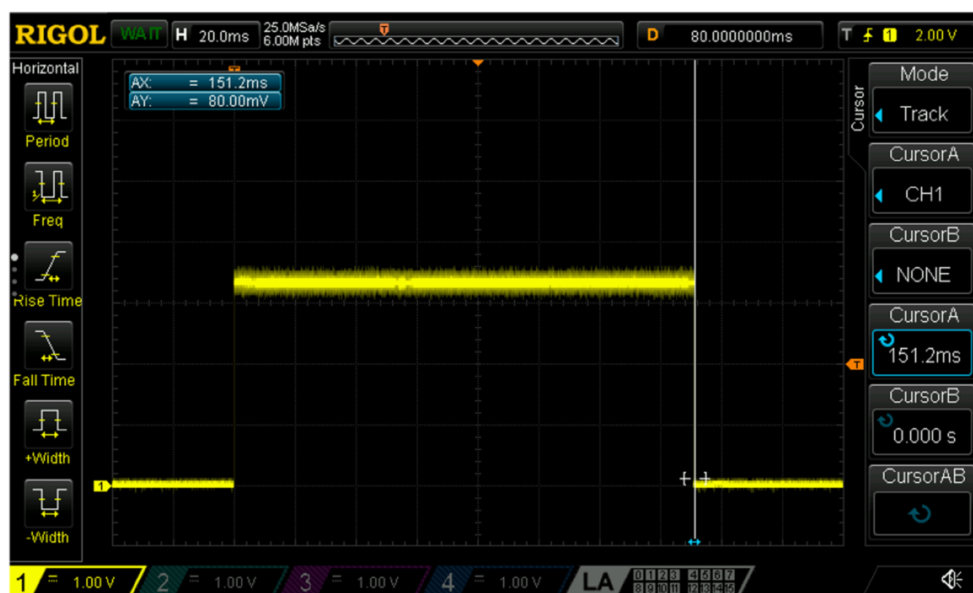


Fig. 7 Time for a scalar multiplication on secp256r1 running on Multos IoT Trust Anchor

Table 10 Internal time spent on scalar multiplication for different curves on Multos IoT Trust Anchor

Private key size (bits)	Curve name	Time (ms)	Mean time (ms)
192	secp192k1	108.2	107.8
	secp192r1	107.4	
256	Wei25519	134.2	147.27
	secp256k1	156.4	
	secp256r1	151.2	
384	secp384r1	285	285
521	secp521r1	485	485

implemented a naive version using `multosMultiply` and `multosDivide`, i.e. it is not optimized and it introduced a limitation on operand size to a maximum of 512 bits. We ran our application on the domain parameters supported for ECC (i.e. scalar multiplication) and for other operations we ran them on operands whose length was of size of N or $2N$ since it is the most common data size used in the studied protocols.

To get the most precise timing for each operation we used a GPIO pin to serve as a trigger and we set it to `HIGH` before the function call and then to `LOW` at the return of the call. As illustrated in Fig. 6, we connected channel 1 of our oscilloscope to the GPIO pin (here GPIO7) used as trigger and to the GND pin. Obviously we checked that the time required for the operations of setting the GPIO pin from `LOW` to `HIGH` and then to `LOW` was negligible.

With this hardware setup and after having arranged the oscilloscope to trigger on a rising edge on channel 1 to start signal capture on this channel, we were able to obtain some measurements like those illustrated in Fig. 7, which is a scalar multiplication on secp256r1.

5.2.3 Performance results on the Multos IoT Trust Anchor

The time measurements we report in Table 10 for the different curves are the time slots measured between the rising edge and the falling edge. This is not exactly an approximation of the scalar multiplication processing time since the API call can certainly contain some additional operations (e.g. checks), but the related time is negligible.

As noted in Table 10 even for domain parameters with the same bit size the scalar multiplication times are different. However for the same curve we always obtained the same results when we multiplied different scalars and points in the scalar multiplication. This is certainly due to the constant time techniques used by the secure OS implementer to avoid information leakage. To make the scalar multiplication measurements comparable to the rest of our measurements presented in Table 11 we decided to use a mean time for a domain parameters size (i.e. private key size).

Table 11 summarizes the results obtained for the different operations and they are reported in Fig. 8.

Figure 8 clearly confirms that the most important (i.e. time consuming) operation to consider in our protocols is scalar multiplication. The curves of performance for each operation are represented with a y-axis using a linear scale on the left and a log scale on the right.

It is worth noting that our results are the first to be published on the Multos IoT Trust Anchor platform.

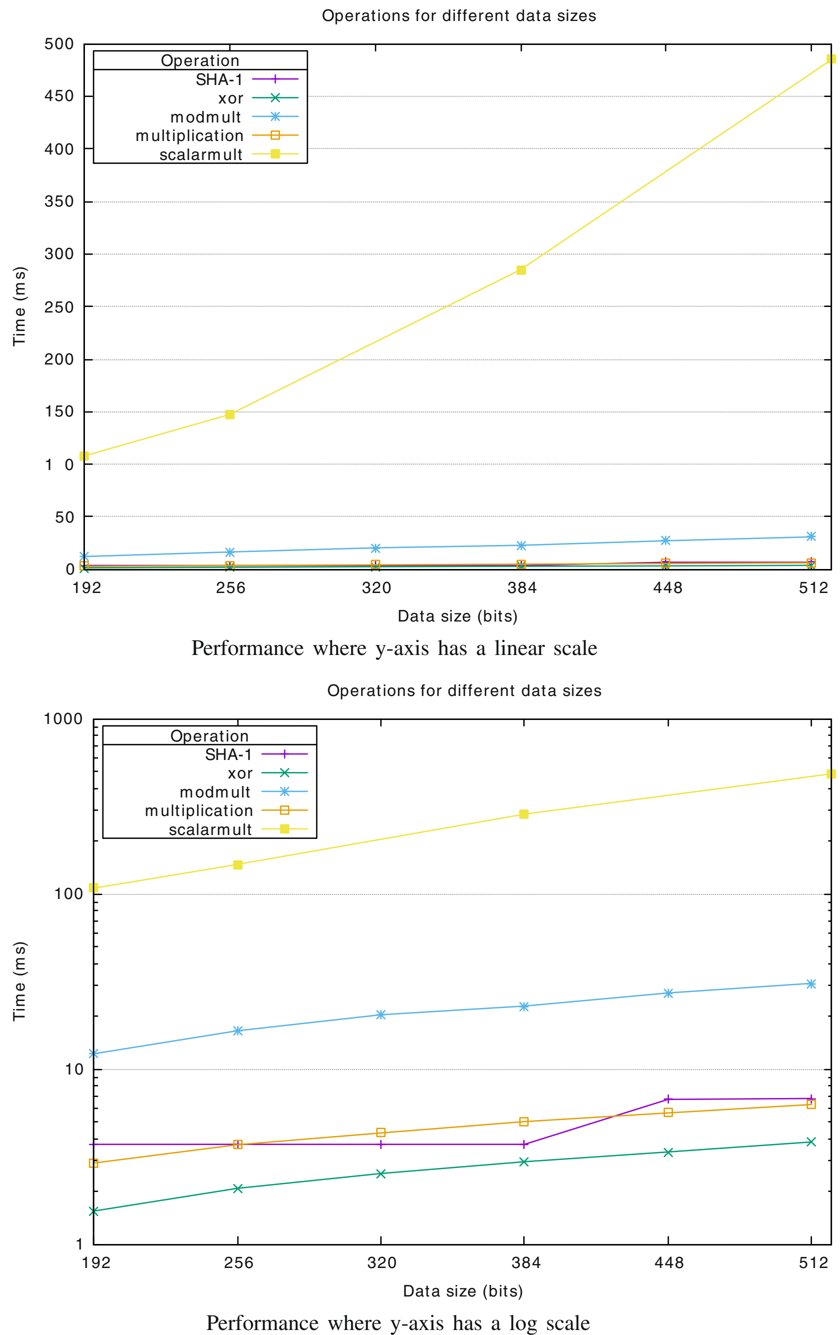
5.2.4 Methodology to measure performance on the Universal JCard

To measure the internal timing of scalar multiplication on the Universal JCard, we needed to use the API calls.

Table 11 Internal time spent to execute the different operations on different data sizes on Multos IoT Trust Anchor (ms)

Operation \ Data size (bits)	192	256	320	384	448	512	521
SHA-1	3.72	3.72	3.72	3.72	6.72	6.80	
xor	1.55	2.09	2.54	2.96	3.37	3.84	
multiplication	2.91	3.70	4.32	5.02	5.63	6.28	
modmult (software implementation)	12.24	16.56	20.44	22.90	27.20	30.90	
scalarmult (mean)	107.8	147.27		285			485

Fig. 8 Performance of different required operations of studied protocols run on Multos IoT Trust Anchor



Performance where y-axis has a linear scale

Performance where y-axis has a log scale

Since Java Card supports Short-Weierstrass curves and the Universal JCard implementation also supports various key sizes up to 521 bits, we did time measurements for several elliptic curves (secp112r1, secp128r1, secp160k1, secp160r1, secp192k1, secp192r1, secp224k1, Wei22519, secp256k1, secp256r1, brainpoolP256r1, secp384r1 and secp521r1).

In short, we implemented a very simple applet creating keys containers, and we sent APDU commands to configure the different domain parameters of these keys. To measure a single scalar multiplication we created a `KeyAgreement` object of type `KeyAgreement.ALG_EC_SVDP_DH_PLAIN`, i.e. the secret value derivation primitive Diffie-Hellmann.

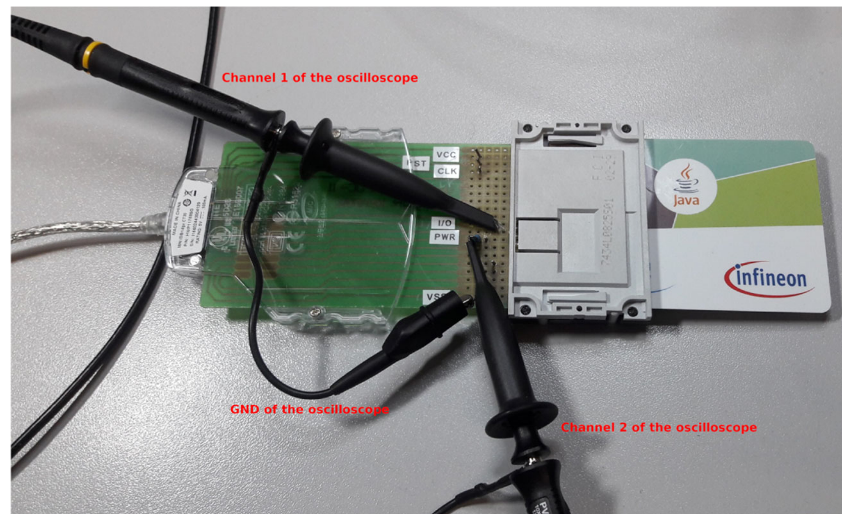
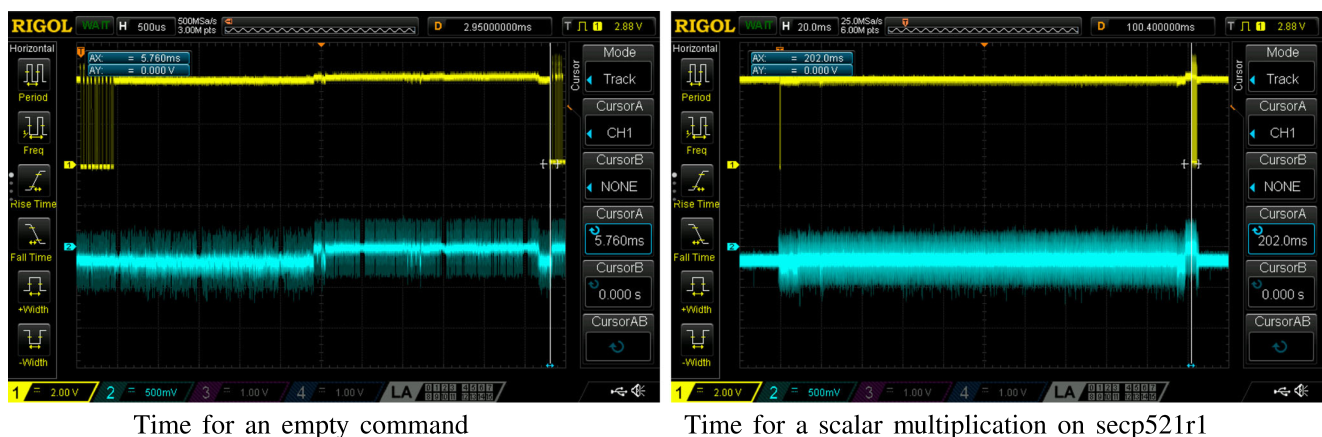


Fig. 9 Universal JCard running our applet and connected to the smart card reader through an adapter enabling measurements with an oscilloscope

We initialised this object with a random private key and requested the secret value derivation by calling `generateSecret` on point P, the generator of the chosen curve: this call is simply a scalar multiplication of the private key (i.e. a scalar) and of P. To avoid inaccuracies due to factors such as overhead and mutex on the computer, the operating system, and delays in communication between computer and smart card, we decided to use an oscilloscope to measure the precise timing between the arrival of the APDU command on the contact smart card interface and the departure of the APDU response for the reader.

As illustrated in Fig. 9, we connected channel 1 of the oscilloscope on I/O line and GND line. We also connected channel 2 to a resistance of 22Ω, which was placed between the GND of the smart card interface and the GND of the reader and reflected the power consumption. The aim of this channel was to check whether it was possible to observe an obvious pattern in power consumption during cryptographic operations, in order to get a better internal timing of scalar multiplication; the expected pattern was not observable (probably due to countermeasures to protect against side-channels).

To avoid any bias, we conducted the experiments for each curve 10 times. This was sufficient since the standard deviation was negligible and we thus kept the average as the result. However, in the code written to perform the required measurement we had some additional instructions, including getting the value of P and copying it to a temporary buffer in RAM. There was also a small overhead on-card due to the APDU dispatcher in the Java Card runtime environment. To get the most accurate measurements of the scalar multiplication, we wanted to measure the time spent to execute all these instructions. We therefore wrote specific code to isolate them, which we called “an empty command” in Fig. 10 enabled us to accurately measure this useless time, based again on the average result of 10 executions.



Time for an empty command

Time for a scalar multiplication on secp521r1

Fig. 10 Performance of different operations run on Universal JCard

Table 12 Internal time spent on scalar multiplication for different curves on Universal JCard

Private key size (bits)	Curve name	Time (ms)	Mean time (ms)
112	secp112r1	28.04	28.04
128	secp128r1	31.24	31.24
160	secp160k1	38.04	37.89
	secp160r1	37.74	
192	secp192k1	45.74	45.49
	secp192r1	45.24	
224	secp224k1	53.94	53.94
256	Wei25519	59.84	62.06
	secp256k1	63.84	
	secp256r1	62.54	
	brainpoolP256r1	62.04	
384	secp384r1	107.04	107.04
521	secp521r1	196.24	196.24

5.2.5 Performance results on the Universal JCard

The time measurements reported in Table 12 are the averages for each curve, from which we subtracted the time spent for the "empty command". This is the best approximation of the time required for a scalar multiplication on each curve.

Table 12 shows that as for the Multos IoT Trust Anchor, for domain parameters with the same bit size, the scalar multiplication times were different. However, for the same curve we always obtained the same results when we multiplied different scalars and points in the scalar multiplication. This was due to the "constant time" techniques used by the secure OS implementer to avoid information leakage. This also explains why only 10 executions for each curve were sufficient, and why the standard deviation was negligible.

To make the scalar multiplication measurements comparable to those obtained for the Multos IoT Trust Anchor, we decided to compute the mean time for a domain parameter size (i.e. private key size) and to illustrate the difference in performances in Fig. 11.

Performances for scalar multiplication on the Universal JCard were much better than those on the Multos

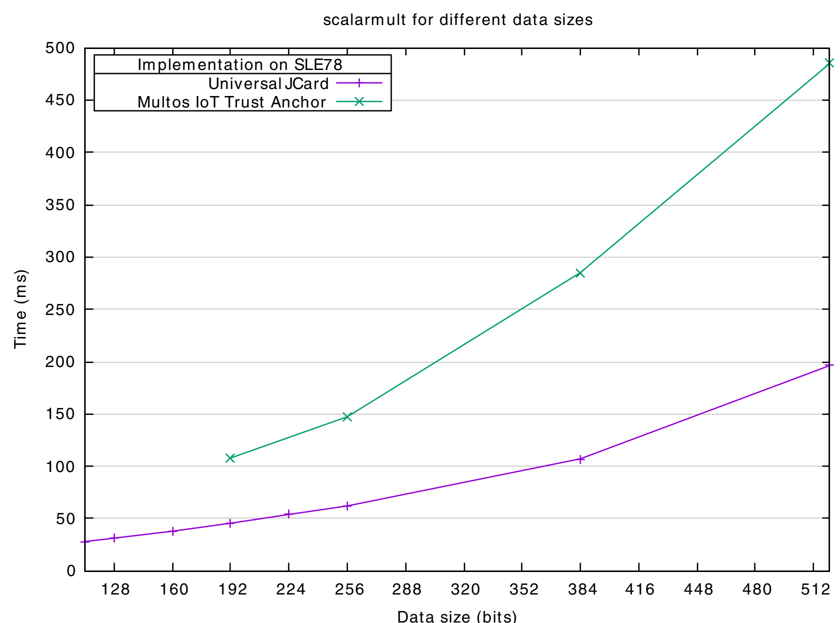


Fig. 11 Comparison of scalar multiplication performance on two SLE78-based devices: Universal JCard and Multos IoT Trust Anchor

Table 13 Estimated computation time required for authentication on smart object side

Protocol	Estimated time (ms)
Our improvements of Jin et al.	186.18
Our hash-based authentication proposal	124.12
Our ECC-based authentication protocol	310.30

IoT Trust Anchor. There may be different explanations, ranging from the internal frequency used (however only the SLE78CLUFX5000PHM may be run at 50MHz, compared with 33MHz for the SLE78CLFX4000P) to library implementations, and OS overheads. Since it was not the aim of this paper to dig into these differences, we simply concluded that scalar multiplication can be achieved efficiently and securely on embedded IoT devices.

5.2.6 Summary

If a 256 bit security is required, and if we consider the scalar multiplication time on the Universal JCard, as0 can be seen in Table 13, the smart object's estimated computation time is reasonable for each of our proposals if other operations are neglected.

6 Conclusion and future work

With the increasing use of IoT technology in healthcare, security and privacy are major concerns, and the cornerstone of security is authentication. Depending on the communication modes, M2C or M2M, the protocols have to be adapted to the context. In this paper, after a study of the state of the art, we established the security and functional requirements and we then proposed three mutual authentication protocols. Two of the three are suitable for the M2C communication mode and the last one is appropriate for the M2M communication mode. Both informal security analysis and formal verification using AVISPA and ProVerif tools have shown they satisfy our security requirements. The performance analysis and the practical experimental results have demonstrated they fulfil the functional requirements. In summary, the three proposed authentication protocols outperform the state-of-the-art systems.

Acknowledgements This work is supported by the ex-Région Limousin, under grant for project "IoTSec", by the MIRES research federation under grants for projects "SCOPE", "SPOCK" and "SPOCK2", by the Région Nouvelle-Aquitaine under the grant for project "SVP-IoT" and by the ID-Fix project, an ANR funded project (ANR-16-CE39-0004).

Appendix: AVISPA and ProVerif specifications for protocols

A.1 AVISPA

A.1.1 Jin et al.'s authentication protocol improvements

There are two basic roles, S and T, which explain the activity of *Server* and T_i.

- There are two agents `T`, `S` and they both use a hash function `H`, a modular multiplication function `MULT`, a scalar multiplication function `MECC`, a modular addition function `ADD`, a points addition function `ADDP` and a curve generator `P`, and a smart object `T` identity `Idt`, its identity hash result `HidT` and server public key `Ps`, to compute the authentication message `AUTHI` and `AUTHS`.
- Secrecy of identity of smart object `T` `Idt`, its identity hash result `HidT`, the server's secret key `Xs` and secret random numbers `Ri`, `Rs` is modeled using the goal predicates `secret(Idt, sec_idt, T)`, `secret(Idt, sec_idt, S)`, `secret(HidT, sec_hidt, T)`, `secret(HidT, sec_hidt, S)`, `secret (Xs, sec_xs , S)`, `secret(Ri', sec_ri, T)`, `secret(Rs', sec_rs, S)` which are maintained by the `protocol_id`: `sec_idt`, `sec_hidt`, `sec_xs`, `sec_ri` and `sec_rs` respectively. The parameters `Idt`, `HidT`, `Xs` and `Ri`, `Rs` are kept secret to `T` and `S`.
- Mutual authentication is achieved via `witness` and `request` goals, i.e. `witness(T, S, auth_i, AUTHI')`, `request(S, T, auth_i, AUTHI')`, `witness(S, T, auth_s, AUTHS')`, `request(T, S, auth_s, AUTHS')`. `witness(T, S, auth_i, AUTHI')` declares that agent `T` claims to be the peer of agent `S`, agreeing on the value `AUTHI'`. `auth_i` is the name of `AUTHI'` authentication shown in the goal section, whereas `request(S, T, auth_i, AUTHI')` declares that agent `S` accepts the value `AUTHI'` and now relies on the guarantee that agent `T` exists and agrees with it on this value.

```
role serverS (S, T:agent,
              P, Xs, Idt, HidT: text,
              Ps: message,
              MECC, MULT, H, ADD, ADDP: hash_func,
              SND, RCV : channel(dy))
played_by   S
def=
local State: nat,
      Rs: text,
      RI, RS, MASK, AUTHS, AUTHI, E, M: message
const sec_rs, sec_xs, sec_idt, sec_hidt, auth_s, auth_i: protocol_id
init State:=0
transition
1.State=0 /\ RCV(start) =|> State':=2 /\ Rs':=new()
                                      /\ RS':=MECC(Rs', P)
                                      /\ secret(Rs', sec_rs, S)
                                      /\ SND(RS')
2.State=2 /\ RCV(RI', AUTHI', M')
          /\ MASK'=MECC(Xs, RI')
          /\ HidT'=xor(M', H(MASK'))
          /\ HidT=H(Idt')
          /\ Idt =xor(AUTHI', H(RS, MASK'))
          /\ Idt'=Idt =|> State':=4 /\ E':=H(RS, RI, Idt)
                                    /\ AUTHS':=MECC(ADD(MULT(Xs, E'), Rs), P)
                                    /\ SND(AUTHS')
                                    /\ secret(Xs, sec_xs, S)
                                    /\ secret(Idt, sec_idt, S)
                                    /\ secret(HidT, sec_hidt, S)
                                    /\ witness(S, T, auth_s, AUTHS')
                                    /\ request(S, T, auth_i, AUTHI')
end role

role soT (T, S: agent,
          P, Idt, HidT: text,
          Ps: message,
          MECC, MULT, H, ADD, ADDP: hash_func,
          SND, RCV: channel(dy))
played_by T def=
local State: nat,
      Ri: text,
      RI, RS, MASK, AUTHS, AUTHI, E, M: message
const sec_ri, sec_idt, sec_hidt, auth_s, auth_i: protocol_id
init State:=1
transition
1.State=1 /\ RCV(RS') =|> State':=3 /\ Ri':=new()
                                    /\ RI':=MECC(Ri', P)
                                    /\ MASK':=MECC(Ri', Ps)
                                    /\ AUTHI':=xor(Idt, H(RS', MASK'))
                                    /\ M':=xor(H(MASK'), HidT)
                                    /\ SND(RI', AUTHI', M')
                                    /\ witness(T, S, auth_i, AUTHI')
                                    /\ secret(Idt, sec_idt, T)
                                    /\ secret(HidT, sec_hidt, T)
                                    /\ secret(Ri', sec_ri, T)
2.State= 3 /\ RCV (AUTHS')
           /\ E'= H(RS, RI, Idt)
           /\ AUTHS'= ADDP(MECC(E',Ps),RS) =|> State':=5 /\ request(T, S, auth_s, AUTHS')
end role

role session(T, S: agent,
             P, Idt, HidT: text,
             Ps: message,
             MECC,MULT, H, ADD, ADDP: hash_func)
def=
local ST, RT, SS, RSS: channel (dy),
      Xs: text
composition
           serverS(S, T, P, Xs, Idt, HidT, Ps, MECC, MULT, H, ADD, ADDP, SS, RSS)
           /\ soT(T, S, P, Idt, HidT, Ps, MECC, MULT, H, ADD, ADDP, ST, RT)
end role
```

Listing 1 AVISPA specification

```
role environment() def=
const  servers, sot, i: agent,
       mecc, mult, h, add, addp: hash_func,
       p, idt, hidt : text,
       ps: message,
       auth_s, auth_i: protocol_id
intruder_knowledge = {servers, sot, i, mecc, mult, h, add, addp}
composition
           session(servers, sot, p, idt, hidt, ps, mecc, mult, h, add, addp)
           /\ session(servers, i, p, idt, hidt, ps, mecc, mult, h, add, addp)
           /\ session(i, sot, p, idt, hidt, ps, mecc, mult, h, add, addp)
end role
goal
secrecy_of sec_ri, sec_rs, sec_xs, sec_idt, sec_hidt
authentication_on auth_s, auth_i
end goal
environment ()
```

Listing 1 (continued)

- The environment contains the global constants and the composition of one or more sessions. The intruder participates as a concrete session in the execution protocol.

A.1.2 Hash-based authentication protocol

There are two basic roles, S and T, which explain the activity of *Server* and T_i.

- There are two agents `T`, `S` and they both use a hash function `H`, a scalar multiplication function `MECC`, a curve generator `P`, and a smart object `T` identity `Idt` and its identity hash result `HidT` respectively to compute the authentication message `AUTHI` and `AUTHS`.
- Secrecy of identity of smart object T `Idt`, its identity hash result `HidT` and secret random numbers `Ri`, `Rs` are modeled using the goal predicates `secret(Idt, sec_idt, T)`, `secret(Idt, sec_idt, S)`, `secret(HidT, sec_hidt, T)`, `secret(HidT, sec_hidt, S)`, `secret(Ri', sec_ri, T)`, `secret(Rs', sec_rs, S)`, which are maintained by the `protocol_id`: `sec_idt`, `sec_hidt`, `sec_ri`, `sec_rs` respectively. The parameters `Idt`, `HidT` and `Ri`, `Rs` are kept secret to `T` and `S`.
- Mutual authentication is achieved via `witness` and `request` goals; i.e., `witness(T, S, auth_i, AUTHI')`, `request(S, T, auth_i, AUTHI')`, `witness (S, T, auth_s, AUTHS')`, `request`

```
role serverS (S, T :agent,
              P, Idt, HidT: text,
              H, MECC: hash_func,
              SND, RCV : channel(dy))
played_by S def=
local State: nat,
      Rs, K:text,
      RI, RS, M, AUTHS, AUTHI: message
      const sec_rs, sec_idt, sec_hidt, auth_i, auth_s: protocol_id
init State:=0
transition
1.State=0 /\ RCV(start) =|> State':=2 /\ Rs':=new()
                                      /\ RS':=MECC(Rs', P)
                                      /\ secret(Rs', sec_rs, S)
                                      /\ SND(RS')
2.State=2 /\ RCV(RI', AUTHI', M')
          /\ K'= MECC(Rs, RI')
          /\ HidT=xor(M', K')
          /\ AUTHI=H(Idt'. 2. K') =|> State':=4 /\ AUTHS':=H(Idt.1.K')
                                                /\ SND(AUTHS')
                                                /\ secret(Idt, sec_idt, S)
                                                /\ secret(HidT, sec_hidt, S)
                                                /\ request( S, T, auth_i, AUTHI')
                                                /\ witness(S, T, auth_s, AUTHS')
end role
```

Listing 2 AVISPA specification

```
role soT(T, S:agent,
         P, Idt, HidT: text,
         H, MECC: hash_func,
         SND, RCV : channel(dy))
played_by T def= local State: nat,
      Ri, K: text,
      RI, RS, M, AUTHS, AUTHI: message
const sec_ri, sec_idt, sec_hidt, auth_i, auth_s : protocol_id
init State:=1
transition
1.State=1 /\ RCV(RS') =|> State':=3 /\ Ri':=new()
                                    /\ RI':=MECC(Ri', P)
                                    /\ K':=MECC(Ri', RS')
                                    /\ AUTHI':=H(Idt.2.K')
                                    /\ M':=xor(HidT, K')
                                    /\ SND(RI', AUTHI', M')
                                    /\ secret(Idt, sec_idt, T)
                                    /\ secret(HidT, sec_hidt, T)
                                    /\ secret(Ri', sec_ri, T)
                                    /\ witness(T, S, auth_i, AUTHI')
2.State=3 /\ RCV(AUTHS')
          /\ AUTHS'=H(Idt.1. K) =|> State':=5 /\ request(T, S, auth_s, AUTHS')
end role

role session(T, S: agent,
             P,Idt, HidT: text,
             H, MECC: hash_func)
def= local ST, RT, SS, RSS: channel(dy) composition
          serverS(S, T, P, Idt, HidT, H, MECC, SS, RSS)
          /\ soT(T, S, P, Idt, HidT, H, MECC, ST, RT)
end role

role environment() def=
const  servers, sot, i: agent,
       h, mecc: hash_func,
       p, idt, hidt: text,
       auth_i, auth_s: protocol_id
intruder_knowledge= {servers, sot, i, h, mecc}
composition
           session(servers, sot, p, idt, hidt, h, mecc)
          /\ session(servers, i, p, idt, hidt, h, mecc)
          /\ session(i, sot, p, idt, hidt, h, mecc)
end role
goal secrecy_of sec_ri, sec_rs, sec_idt, sec_hidt
authentication_on auth_i, auth_s
end goal
environment ()
```

Listing 2 (continued)

(T, S, auth_s, AUTHS'). witness(T, S, auth_i, AUTHI') declares that agent T claims to be the peer of agent S, agreeing on the value AUTHI'. auth_i is the name of AUTHI' authentication shown in the goal section whereas request(S, T, auth_i, AUTHI') declares that agent S accepts the value AUTHI' and now relies on the guarantee that agent T exists and agrees with it on this value.

– The environment contains the global constants and the composition of one or more sessions. The intruder participates as a concrete session in the execution protocol.

A.1.3 ECC-based authentication protocol

There are two basic roles, A and B, which explain the activity of T_i and T_j nodes.

– There are two agents A, B and they both use a hash function H, a modular multiplication function MULT, a scalar multiplication function MECC, a points addition function ADDP, an initialization function INITVAR and a curve generator P, server public key Ps, private/public keys Xa/Pa, Xb/Pb, and public key Schnorr signature (Ppa, Za), (Ppb, Zb) respectively to

```
role soA(A,B: agent,
         Xa, P: text,
         Ps, Pa, PPa, Za: message,
         H, MULT, MECC, ADDP, INITVAR: hash_func,
         SND, RCV: channel(dy))
played_by A def=
local State: nat,
      Ra: text,
      R1, R2, AUTHA, AUTHB, Pb, PPb, Zb, INTER: message
const sec_xa, sec_ra, auth_a, auth_b: protocol_id
init State:=0
transition
1.State=0 /\ RCV(start) =|> State':=2 /\ Ra':=new()
                                      /\ R1':=MECC(Ra', Pa)
                                      /\ secret(Xa, sec_xa, A)
                                      /\ secret(Ra', sec_ra, A)
                                      /\ SND(R1', Pa, PPa, Za)

2. State= 2 /\ RCV(R2', AUTHB', Pb', PPb', Zb')
            /\ INTER'=INITVAR
            /\ INTER'=MECC(H(Pb',PPb'), Ps)
            /\ ADDP(PPb', INTER')=MECC(Zb', P)
            /\ AUTHB' = MECC(MULT(Ra, Xa), R2') =|> State':=4 /\ AUTHA':=MECC(Xa, R2')
                                                               /\ SND(AUTHA')
                                                               /\ request(A, B, auth_b, AUTHB')
                                                               /\ witness(A, B, auth_a, AUTHA')
end role

role soB (B, A: agent,
          Xb, P: text,
          Ps, Pb, PPb, Zb: message,
          H, MULT, MECC, ADDP, INITVAR: hash_func,
          SND, RCV: channel(dy))
played_by B def=
local State: nat,
      Rb: text,
      R2, R1, AUTHB, AUTHA, Pa, PPa, Za, INTER: message
const sec_xb, sec_rb,auth_a, auth_b: protocol_id
init State:=1
transition 1.State=1 /\ RCV(R1', Pa', PPa', Za')
          /\ INTER'=INITVAR
          /\ INTER'=MECC(H(Pa', PPa'), Ps)
          /\ ADDP(PPa', INTER')=MECC(Za', P) =|> State':=3 /\ Rb':=new()
                                                          /\ R2':=MECC(Rb', Pb)
                                                          /\ AUTHB':=MECC(Xb, R1')
                                                          /\ secret(Xb, sec_xb, B)
                                                          /\ secret(Rb', sec_rb, B)
                                                          /\ SND(Pb, R2', AUTHB', PPb, Zb)
                                                          /\ witness(B, A, auth_b, AUTHB')
2.State=3 /\ RCV(AUTHA')
          /\ AUTHA'=MECC(MULT(Rb, Xb), R1') =|> State':=5 /\ request(B, A, auth_a, AUTHA')
end role

role session(A, B: agent,
            P: text,
            Ps: message,
            H, MULT, MECC, ADDP, INITVAR: hash_func)
def=
local SA, RA, SB, RB: channel(dy),
      XA, XB : text,
      Pa, Pb, PPa, Za, PPb, Zb: message
composition
          soA(A, B, P, XA, Ps, Pa, PPa, Za, H, MULT, MECC, ADDP, INITVAR, SA, RA)
          /\ soB(B, A, P, XB, Ps, Pb, PPb, Zb, H, MULT, MECC, ADDP, INITVAR, SB, RB)
end role

role environment() def=
const soa, sob, i: agent,
      pa, pb, pi, ps, ppa, za, ppb, zb: message,
      h, mult, mecc, addp, initvar: hash_func,
      xa, xb, p: text,
```

Listing 3 AVISPA specification

Listing 3 (continued)

```
      auth_a, auth_b: protocol_id
intruder_knowledge={soa, sob, i, h, mult, mecc, addp, initvar}
composition
          session(soa, sob, p, ps, h, mult, mecc, addp, initvar)
          /\ session(soa, i, p, ps, h, mult, mecc, addp, initvar)
          /\ session (i, sob, p, ps, h, mult, mecc, addp, initvar)
end role
goal
secrecy_of sec_xa, sec_xb, sec_ra, sec_rb
authentication_on auth_a, auth_b
end goal
environment()
```

compute the authentication messages AUTHA and AUTHB.

- Secrecy of private keys Xa, Xb and secret random numbers Ra, Rb is modeled using the goal predicates secret(Xa, sec_xa, A), secret(Xb, sec_xb, B), secret(Ra', sec_ra, A), secret (Rb', sec_rb, B), which are maintained by the protocol_id: sec_xa, sec_xb, sec_ra, sec_rb respectively. The parameters (Xa, Ra) and (Xb, Rb) are kept secret to A and B respectively.
- Mutual authentication is achieved via witness and request goals i.e. witness(A, B, auth_a, AUTHA'), request(B, A, auth_a, AUTHA'), witness(B, A, auth_b, AUTHB'), request(A, B, auth_b, AUTHB'). witness(A, B, auth_a, AUTHA') declares that agent A claims to be the peer of agent B, agreeing on the value AUTHA'. auth_a is the name of AUTHA' authentication shown in the goal section whereas request(B, A, auth_a, AUTHA') declares that agent B accepts the value AUTHA' and now relies on the guarantee that agent A exists and agrees with it on this value.
- The environment contains the global constants and the composition of one or more sessions. The intruder participates as a concrete session in the execution protocol.

A.2 ProVerif

A.2.1 Applied Pi calculus specification script of Jin et al.'s authentication protocol improvements

Explanations of the applied pi calculus scripts are the following:

- The secrets xs, idt and hidt are declared as secret to the attacker using the word [private]. ch is the public channel where SERVERS and SOI exchange their messages. The one-way hash function is modeled by H1, H2 and H3, for hashing one, two and three elements respectively. mult, mecc, add, addp, concat and xor represent modular multiplication, scalar multiplication, modular addition, points addition, concatenation and xor functions respectively.
- Secrecy of idt, hidt, xs, ri and rs is verified with queries query attacker(idt), query attacker(hidt), query attacker(xs), query attacker(new ri) and query attacker(new rs).

Listing 4 ProVerif specification

```
   (* Jin et al.'s protocol improvements *)

free ch: channel.
free xs, idt, hidt: bitstring [private].
free ps, p: bitstring.
fun H1(bitstring):bitstring.
fun H2(bitstring, bitstring): bitstring.
fun H3(bitstring, bitstring, bitstring): bitstring.
fun mult(bitstring, bitstring): bitstring.
fun mecc(bitstring, bitstring): bitstring.
fun add(bitstring, bitstring): bitstring.
fun addp(bitstring, bitstring):
bitstring. fun xor(bitstring,bitstring): bitstring.
equation forall x: bitstring, y: bitstring; xor(xor(x, y), y)=x.

(* Queries *)
(* verify secrets *)
query attacker(xs).
query attacker(idt).
query attacker(hidt).
query attacker(new ri).
```

```
query attacker(new rs).
(* verify authentication *)
event beginT(bitstring).
event endT(bitstring).
event beginS(bitstring).
event endS(bitstring).
query x: bitstring; inj-event(beginS(x)) ==> inj-event(beginT(x)).
query x: bitstring; inj-event(endT(x)) ==> inj-event(endS(x)).
query x: bitstring, y:bitstring; inj-event(endS(y)) ==> inj-event(beginS(x))&&inj-event (beginT(x)).

(* Role of the Server S *)
let SERVERS=
new rs: bitstring;
let RS=mecc(rs, p) in
out(ch, (RS));
in(ch,(m2:bitstring, m3:bitstring, m4:bitstring));
let Mask=mecc(xs, m2) in
let hidtprime=xor(m4, H1(Mask)) in
let idtprime=xor(m3, H2(RS, Mask)) in
if idt=idtprime then
let e= H3(RS, m2, idt) in
let Auths= mecc(add(mult(xs, e), rs), p) in
event beginS(Auths);
out (ch, Auths);
event endS(Auths).

(* Role of the Smart Object I *)
let SOI=
in(ch,(m1: bitstring));
new ri: bitstring;
let RI=mecc(ri, p) in
let Mask=mecc(ri, ps) in
let Authi=xor(idt, H2(m1, Mask)) in
let M=xor(H1(Mask), hidt) in
event beginT(Authi);
out(ch, (RI, Authi, M));
in(ch, m5: bitstring);
let e=H3(m1, RI, idt) in
if m5=addp(mecc(e, ps), m1)then
event endT(Authi).

(* Start process *)
process
((* Launch an unbounded number of sessions of the smart object AND server *)
   (!SERVERS) | (!SOI)
```

Listing 4 (continued)

- Mutual authentication between the T_i and Server is modeled with the definition of four events that are mapped in the SOI and SERVERS sub-processes and the following queries:

```
event beginT(bitstring).
event endT(bitstring).
event beginS(bitstring).
event endS(bitstring).

query x: bitstring; inj-event(beginS(x))
         ==> inj-event(beginT(x)).
query x: bitstring; inj-event(endT(x))
         ==> inj-event(endS(x)).
query x: bitstring, y:bitstring;
         inj-event(endS(y))
         ==> inj-event(beginS(x))
         && inj-event(beginT(x)).
```

- In the main process, SERVERS and SOI sub-processes are running in parallel. ! indicates an unlimited number of processes:

```
process
((!SERVERS) | (!SOI))
```

A.2.2 ProVerif results of Jin et al.'s authentication protocol improvements

A.2.3 Applied Pi calculus specification script of hash-based authentication protocol

Explanations of the applied pi calculus scripts are the following:

```
-- Query not attacker(xs[])
Completing...
Starting query not attacker(xs[])
RESULT not attacker(xs[]) is true.
-- Query not attacker(idt[])
Completing...
Starting query not attacker(idt[])
RESULT not attacker(idt[]) is true.
-- Query not attacker(hidt[])
Completing...
Starting query not attacker(hidt[])
RESULT not attacker(hidt[]) is true.
-- Query not attacker(ri[m1 = v_2207,!1 = v_2208])
Completing...
Starting query not attacker(ri[m1 = v_2207,!1 = v_2208])
RESULT not attacker(ri[m1 = v_2207,!1 = v_2208]) is true.
-- Query not attacker(rs[!1 = v_2765])
Completing...
Starting query not attacker(rs[!1 = v_2765])
RESULT not attacker(rs[!1 = v_2765]) is true.
-- Query inj-event(beginS(x_30)) ==> inj-event(beginT(x_30))
Completing...
Starting query inj-event(beginS(x_30)) ==> inj-event(beginT(x_30))
RESULT inj-event(beginS(x_30)) ==> inj-event(beginT(x_30)) is true.
-- Query inj-event(endT(x_31)) ==> inj-event(endS(x_31))
Completing...
Starting query inj-event(endT(x_31)) ==> inj-event(endS(x_31))
RESULT inj-event(endT(x_31)) ==> inj-event(endS(x_31)) is true.
-- Query inj-event(endS(y_33)) ==> (inj-event(beginS(x_32)) && inj-event(beginT(x_32)))
Completing...
Starting query inj-event(endS(y_33)) ==> (inj-event(beginS(x_32)) && inj-event(beginT(x_32)))
RESULT inj-event(endS(y_33)) ==> (inj-event(beginS(x_32)) && inj-event(beginT(x_32))) is true.
```

Listing 5 ProVerif results

```
    (* Hash-based authentication protocol *)
free ch: channel.
free idt, hidt: bitstring [private].
free ps, p: bitstring.
const v1,v2: bitstring.
fun h (bitstring): bitstring [data].
fun mecc (bitstring, bitstring):bitstring.
fun concat (bitstring, bitstring):bitstring.
fun xor (bitstring, bitstring): bitstring.
equation forall x:bitstring, y: bitstring; xor(xor(x,y),y)=x.

(* Queries *)
(* verify secrets*)
query attacker(idt).
query attacker(hidt).
query attacker(new ri).
query attacker(new rs).
(* verify authentication *)
event beginT(bitstring).
event endT(bitstring).
event beginS(bitstring).
event endS(bitstring).
query x: bitstring; inj-event(beginS(x)) ==> inj-event(beginT(x)).
query x: bitstring; inj-event(endT(x)) ==> inj-event(endS(x)).
query x: bitstring, y:bitstring; inj-event(endS(y)) ==> inj-event(beginS(x)) && inj-event(beginT(x)).

(* Role of the Server S *)
let SERVERS =
new rs:bitstring;
let RS=mecc(rs, p) in
out(ch, (RS));
in(ch,(m2: bitstring, m3: bitstring, m4: bitstring));
let KS=mecc(rs, m2) in
let hidtprime = xor(m4, KS) in
if hidt=hidtprime then
if m3=h(concat(idt, concat(v2, KS))) then
let AUTHS=h(concat(idt, concat(v1, KS))) in
event beginS(AUTHS);
```

Listing 6 ProVerif specification

```
out (ch, AUTHS);
event endS(AUTHS).

(* Role of the smart object I *)
let SOI =
in(ch,(m1: bitstring));
new ri: bitstring;
let RI=mecc(ri, p) in
let KI=mecc(ri, m1) in
let h(=idt)=hidt in
let M=xor(hidt, KI) in
let AUTHI=h(concat(idt, concat(v2, KI))) in
event beginT(AUTHI);
out(ch,(RI, AUTHI, M));
in(ch, m5: bitstring);
if m5= h(concat(idt, concat(v1, KI)))then
event endT(AUTHI).

(* Start process *)
process
    ((* Launch an unbounded number of sessions of the TAG AND SERVER*)
          (!SERVERS) | (!SOI))
```

Listing 6 (continued)

- The secrets `idt`, `hidt` are declared as secret to the attacker using the word `[private]`. `ch` is the public channel where `SERVERS` and `SOI` exchange their messages and `h`, `mecc`, `concat`, `xor` represent hash, scalar multiplication, concatenation and xor functions respectively.
- Secrecy of `idt`, `hidt`, `ri` and `rs` is verified with queries `query attacker(idt)`, `query attacker(hidt)`, `query attacker(new ri)` and `query attacker(new rs)`.
- Mutual authentication between the T_i and Server is modeled with definition of four events that are mapped in the `SOI` and `SERVERS` sub-processes and the following queries:

```
event beginT(bitstring).
event endT(bitstring).
event beginS(bitstring).
event endS(bitstring).

query x: bitstring; inj-event(beginS(x))
         ==> inj-event(beginT(x)).
query x: bitstring; inj-event(endT(x))
         ==> inj-event(endS(x)).
query x: bitstring, y:bitstring;
         inj-event(endS(y))
         ==> inj-event(beginS(x))
         && inj-event(beginT(x)).
```

- In the main process, `SERVERS` and `SOI` sub-processes are running in parallel. `!` indicates an unlimited number of processes:

```
process
 ((!SERVERS) | (!SOI))
```

A.2.4 ProVerif results of hash-based authentication protocol

A.2.5 Applied Pi calculus specification script of ECC-based authentication protocol

Explanations of the applied pi calculus scripts are the following:

- The secrets `xa`, `xb` are declared as secret to the attacker using the word `[private]`. `ch` is the public channel where `SOA` and `SOB` exchange their messages and `h`, `mult`, `mecc`, `addp` represent hash, multiplication, scalar multiplication and points addition functions respectively.
- Secrecy of `xa`, `xb`, `ra` and `rb` is verified with `query attacker(xa)`, `query attacker(xb)`, `query attacker(new ra)` and `query attacker(new rb)`.
- Mutual authentication between T_i and T_j is modeled with the definition of four events that are mapped in the `SOA` and `SOB` sub-processes and the following queries:

```
event beginA(bitstring).
event endA(bitstring).
event beginB(bitstring).
event endB(bitstring).

query x: bitstring; inj-event(beginB(x))
         ==> inj-event(beginA(x)).
query x: bitstring; inj-event(endA(x))
         ==> inj-event(endB(x)).
query x: bitstring, y:bitstring;
         inj-event(endA(y))
         ==> inj-event(beginA(x))
         && inj-event(beginB(x)).
```

```
-- Query not attacker(idt[])
Completing...
Starting query not attacker(idt[])
RESULT not attacker(idt[]) is true.
-- Query not attacker(hidt[])
Completing...
Starting query not attacker(hidt[])
RESULT not attacker(hidt[]) is true.
-- Query not attacker(ri[m1 = v_766,!1 = v_767])
Completing...
Starting query not attacker(ri[m1 = v_766,!1 = v_767])
RESULT not attacker(ri[m1 = v_766,!1 = v_767]) is true.
-- Query not attacker(rs[!1 = v_1021])
Completing...
Starting query not attacker(rs[!1 = v_1021])
RESULT not attacker(rs[!1 = v_1021]) is true.
-- Query inj-event(beginS(x_28)) ==> inj-event(beginT(x_28))
Completing...
Starting query inj-event(beginS(x_28)) ==> inj-event(beginT(x_28))
RESULT inj-event(beginS(x_28)) ==> inj-event(beginT(x_28)) is true.
-- Query inj-event(endT(x_29)) ==> inj-event(endS(x_29))
Completing...
Starting query inj-event(endT(x_29)) ==> inj-event(endS(x_29))
RESULT inj-event(endT(x_29)) ==> inj-event(endS(x_29)) is true.
-- Query inj-event(endS(y_31)) ==> (inj-event(beginS(x_30)) && inj-event(beginT(x_30)))
Completing...
Starting query inj-event(endS(y_31)) ==> (inj-event(beginS(x_30)) && inj-event(beginT(x_30)))
RESULT inj-event(endS(y_31)) ==> (inj-event(beginS(x_30)) && inj-event(beginT(x_30))) is true.
```

Listing 7 ProVerif results

```
    (* ECC based authentication protocol *)

free ch: channel.
free xa, xb: bitstring [private].
free pa, pb, ps, p, ppa, ppb, za, zb: bitstring.
fun h (bitstring, bitstring): bitstring.
fun mult (bitstring, bitstring):bitstring.
fun mecc (bitstring, bitstring):bitstring.
fun addp (bitstring, bitstring):bitstring.

(* Queries *)
(* verify secrets *)
query attacker(xa).
query attacker(xb).
query attacker(new ra).
query attacker(new rb).
(* verify authentication *)
event beginA(bitstring).
event endA(bitstring).
event beginB(bitstring).
event endB(bitstring).
query x: bitstring; inj-event(beginB(x)) ==> inj-event(beginA(x)).
query x: bitstring; inj-event(endA(x)) ==> inj-event(endB(x)).
query x: bitstring, y:bitstring; inj-event(endA(y)) ==> inj-event(beginA(x)) && inj-event(beginB(x)).

(* Role of the smart object A *)
let SOA=
new ra: bitstring;
let RA=mecc(ra, pa) in
out(ch, (RA, pa, ppa, za));
in(ch,(m5: bitstring, m6: bitstring, m7: bitstring, m8: bitstring, m9: bitstring));
if addp(mecc(h(m7, m8), ps), m8)= mecc(m9, p) then
if m6=mecc( mult(ra, xa), m7) then
let AUTHA=mecc(xa, m5) in
event beginA(AUTHA);
out (ch, AUTHA);
event endA(AUTHA).
```

Listing 8 ProVerif specification

```
(* Role of the smart object B *)
let SOB=
in(ch,(m1: bitstring, m2: bitstring, m3: bitstring, m4: bitstring));
new rb: bitstring;
if addp(mecc(h(m2, m3), ps), m3)=mecc(m4, p) then
let RB= mecc(rb, pb) in
let AUTHB=mecc(xb, m1) in
event beginB(AUTHB);
out(ch,(RB, AUTHB, pb, ppb, zb));
in(ch, m10: bitstring);
if m10= mecc( mult(rb, xb),m2) then
event endB(AUTHB).

(* Start process *)
process((* Launch an unbounded number of sessions of the SOA AND SOB *)
          (!SOA) | (!SOB))
```

Listing 8 (continued)

– In the main process, SOA and SOB sub-processes are running in parallel. ! indicates an unlimited number of processes:

```
process
((!SOA) | (!SOB))
```

A.2.6 ProVerif results of ECC-based authentication protocol

```
-- Query not attacker(xa[])
Completing...
Starting query not attacker(xa[])
RESULT not attacker(xa[]) is true.
-- Query not attacker(xb[])
Completing...
Starting query not attacker(xb[])
RESULT not attacker(xb[]) is true.
-- Query not attacker(ra[!1 = v_931])
Completing...
Starting query not attacker(ra[!1 = v_931])
RESULT not attacker(ra[!1 = v_931]) is true.
-- Query not attacker(rb[m4 = v_1234,m3 = v_1235,m2 = v_1236,m1 = v_1237,!1 = v_1238])
Completing...
Starting query not attacker(rb[m4 = v_1234,m3 = v_1235,m2 = v_1236,m1 = v_1237,!1 = v_1238])
RESULT not attacker(rb[m4 = v_1234,m3 = v_1235,m2 = v_1236,m1 = v_1237,!1 = v_1238]) is true.
-- Query inj-event(beginB(x)) ==> inj-event(beginA(x))
Completing...
Starting query inj-event(beginB(x)) ==> inj-event(beginA(x))
RESULT inj-event(beginB(x)) ==> inj-event(beginA(x)) is true.
-- Query inj-event(endA(x_12)) ==> inj-event(endB(x_12))
Completing...
Starting query inj-event(endA(x_12)) ==> inj-event(endB(x_12))
RESULT inj-event(endA(x_12)) ==> inj-event(endB(x_12)) is true.
-- Query inj-event(endA(y)) ==> (inj-event(beginA(x_13)) && inj-event(beginB(x_13)))
Completing...
Starting query inj-event(endA(y)) ==> (inj-event(beginA(x_13)) && inj-event(beginB(x_13)))
RESULT inj-event(endA(y)) ==> (inj-event(beginA(x_13)) && inj-event(beginB(x_13))) is true.
```

Listing 9 ProVerif results

References

1. Atzori L, Iera A, Morabito G (2017) Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm. Ad Hoc Networks 56:122–140
2. Sathish Kumar J, Patel DR (2014) A survey on internet of things: security and privacy issues. Int J Comput Appl 90(11):20–26
3. Hail MA, Fischer S (2015) Iot for aal: an architecture via information-centric networking. In: Globecom workshops (GC Wkshps), 2015 IEEE. IEEE, pp 1–6
4. Sicari S, Rizzardi A, Grieco LA, Coen-Porisini A (2015) Security, privacy and trust in internet of things: the road ahead. Comput Netw 76:146–164
5. Lu R, Li X, Liang X, Shen X, Lin X (2011) Grs: the green, reliability, and security of emerging machine to machine communications. IEEE communications magazine 49(4):28–35
6. Saied YB, Olivereau A, Laurent M (2012) A distributed approach for secure m2m communications. In: 2012 5th international conference on new technologies, mobility and security (NTMS). IEEE, pp 1–7
7. Ren W, Yu L, Ma L, Ren Y (2013) Rise: a reliable and secure scheme for wireless machine to machine communications. Tsinghua Sci Technol 18(1):100–117
8. Domingo MC (2012) An overview of the internet of things for people with disabilities. J Netw Comput Appl 35(2):584–596
9. Porambage P, Braeken A, Gurtov A, Ylianttila M, Spinsante S (2015) Secure end-to-end communication for constrained devices in IoT-enabled ambient assisted living systems. In: IEEE 2nd world forum on internet of things (WF-IoT). IEEE, p 2015
10. Shamir A (1979) How to share a secret. Commun ACM 22(11):612–613
11. Nguyen KT, Oualha N, Laurent M (2016) Authenticated key agreement mediated by a proxy re-encryptor for the internet of things. In: European symposium on research in computer security. Springer, pp 339–358
12. Nuñez D, Agudo I, Lopez J (2016) Attacks to a proxy-mediated key agreement protocol based on symmetric encryption. IACR Cryptology ePrint Archive 2016:1081
13. Jin C, Xu C, Zhang X, Li F (2016) A secure ecc-based rfid mutual authentication protocol to enhance patient medication safety. J Med Syst 40(1):12
14. Zhao Z (2014) A secure rfid authentication protocol for healthcare environments using elliptic curve cryptosystem. J Med Syst 38(5):46
15. Alamr AA, Kausar F, Kim J, Seo C (2018) A secure ecc-based rfid mutual authentication protocol for internet of things. J Supercomput 74:4281–4294
16. Liao Y-P, Hsiao C-M (2013) A secure ecc-based rfid authentication scheme using hybrid protocols. In: Advances in intelligent systems and applications, vol 2. Springer, pp 1–13
17. Dinarvand N, Barati H (2019) An efficient and secure rfid authentication protocol using elliptic curve cryptography. Wirel Netw 25(1):415–428
18. Sharma D, Jinwala D (2015) Functional encryption in IoT e-health care system. In: International conference on information systems security. Springer, pp 345–363
19. Li R, Shen C, He H, Xu Z, Xu C-Z (2017) A lightweight secure data sharing scheme for mobile cloud computing. IEEE Transactions on Cloud Computing 6:344–357
20. Porambage P, Braeken A, Kumar P, Gurtov A, Ylianttila M (2015) Proxy-based end-to-end key establishment protocol for the internet of things. In: IEEE international conference on communication workshop (ICCW). IEEE, p 2015
21. Amin R, Hafizul Islam SK, Biswas GP, Khan MK, Kumar N (2018) A robust and anonymous patient monitoring system using wireless medical sensor networks. Futur Gener Comput Syst 80:483–495
22. Jiang Q, Ma J, Yang C, Ma X, Shen J, Chaudhry SA (2017) Efficient end-to-end authentication protocol for wearable health monitoring systems. Comput Electr Eng 63:182–195
23. Tuna G, Kogias DG, Cagri Gungor V, Gezer C, Takn E, Ayday E (2017) A survey on information security threats and solutions for machine to machine (m2m) communications. J Parallel Distrib Comput 109:142–154
24. Saadeh M, Sleit A, Qatawneh M, Almobaideen W (2016) Authentication techniques for the internet of things: a survey. In: 2016 cybersecurity and cyberforensics conference (CCC), pp 28–34
25. Ferrag MA, Maglaras LA, Janicke H, Jiang J, Shu L (2017) Authentication protocols for internet of things: a comprehensive survey. Security and Communication Networks, 2017
26. Dolev D, Yao A (1983) On the security of public key protocols. IEEE Trans Inf Theory 29(2):198–208
27. TA Team et al (2006) Avispa v1. 1 user manual. Information society technologies programme (June 2006), http://avispa-project.org
28. Blanchet B (2014) Automatic verification of security protocols in the symbolic model: the verifier proverif. In: Foundations of security analysis and design VII. Springer, pp 54–87
29. Bonnefoi P-F, Dusart P, Sauveron D, Akram RN, Markantonakis K (2015) A set of efficient privacy protection enforcing lightweight authentication protocols for low-cost rfid tags. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol 1. IEEE, pp 612–620
30. Zhuang Y, Yang A, Hancke GP, Wong DS, Yang G (2018) Energy-efficient distance-bounding with residual charge computation. IEEE Trans Emerg Top Comput, pp 1–1
31. Ong H, Schnorr C-P, Shamir A (1984) An efficient signature scheme based on quadratic equations. In: Proceedings of the sixteenth annual ACM symposium on theory of computing. ACM, pp 208–216
32. MULTOS Limited (2019) Multos Trust Anchor. https://www.multos.com/trust_anchor. Online; accessed 20 May 2019
33. Universal Smart Cards Limited (2019) Universal JCard. http://www.universaljcard.com/. Online; accessed 20 May 2019
34. MULTOS Limited (2019) Multos consortium. https://www.multos.com/consortium. Online; accessed 20 May 2019
35. Infineon Technologies AG (2019) SLE 78CLUFX5000PH. https://www.infineon.com/cms/en/product/security-smart-card-solutions/security-controllers-for-usb-tokens/sle-78clufx5000ph/. Online; accessed 20 May 2019
36. Infineon Technologies AG (2019) SLE 78CLFX4000P. https://www.infineon.com/cms/en/product/security-smart-card-solutions/security-controllers/sle-78/sle-78clfx4000p/. Online; accessed 20 May 2019
37. Van haver P (2019) Unveiling Java Card 3.1: new I/O and trusted peripherals. https://blogs.oracle.com/javaiot/unveiling-java-card-31%3a-new-io-and-trusted-peripherals, November 2018. Online; accessed 20 May 2019
38. Oracle Inc (2019) Oracle Java Card boosts security for IoT devices at the edge. https://www.oracle.com/corporate/pressrelease/oracle-java-card-boosts-security-011619.html, January 2019. Online; accessed 20 May 2019
39. Oracle Inc. (2019) Java Card 3.1 documentation. https://docs.oracle.com/en/java/javacard/3.1/, January 2019. Online; accessed 20 May 2019
40. Infineon Technologies AG (2019) Integrity guard. https://www.infineon.com/dgdl/infineon-integrity_guard_the_smartest_digital_security_technology_in_the_industry_06.18-WP-v01_01-EN.pdf?fileid=5546d46255dd933d0155e31c46fa03fb, June 2018. Online; accessed 20 May 2019

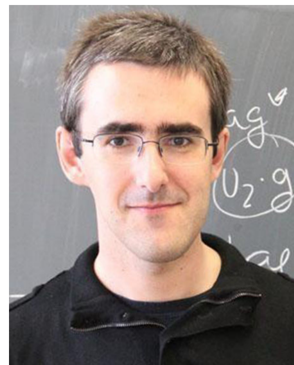
41. Infineon Technologies AG (2019) SOLID FLASH chip card controllers. https://www.infineon.com/dgdl/infineon-SOLID_FLASH_chip_card_controllers-PB-v04_12-EN.pdf?fileid=5546d46255dd933d0155e31d0ac105d9, May 2012. Online; accessed 20 May 2019
42. MULTOS Limited (2019) Multos IoT Trust Anchor Developer Boards. https://www.multos.com/dev_boards. Online; accessed 20 May 2019
43. MULTOS Limited (2019) ML5-P22 and MC5-p22 on INFINEON SLE78 PLATFORM. https://www.multos.com/products/approved_platforms/MIR/multos_international/m5-p22, Online; accessed 20 May 2019
44. MULTOS Limited (2019) ML5-P19 and MC5-p19 on INFINEON SLE78 PLATFORM. https://www.multos.com/products/approved_platforms/MIR/multos_international/m5_p19. Online; accessed 20 May 2019
45. MULTOS Limited (2019) Smartdeck. https://www.multos.com/software, Online; accessed 20 May 2019
46. Oracle Inc. (2019) Java Card technology. https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html. Online; accessed 20 May 2019
47. Sauveron D (2009) Multiapplication smart card: towards an open smart card? Inf Secur Tech Rep 14(2):70–78. Smart Card Applications and Security
48. Infineon Technologies AG (2019) Java Card IDE. https://www.infineon.com/cms/en/product/promopages/devkit4ID/. Online; accessed 20 May 2019
49. Oracle Inc. (2019) Java Card Development Kit. https://www.oracle.com/technetwork/java/embedded/javacard/downloads/javacard-sdk-2043229.html. Online; accessed 20 May 2019

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Fatma Merabet is a PhD student in LARI laboratory from Mouloud Mammeri University of Tizi Ouzou (UMMTO, Algeria) and XLIM laboratory (UMR CNRS 7252/Université de Limoges, France). Currently, she focuses her research activities on authentication protocols and short range wireless technologies for IoT around the topic: "IoT Security Solutions for Ambient Assisted Living".

Amina Cherif received the Master degree in Networks, Mobility and Embedded Systems from Mouloud Mammeri University of Tizi Ouzou (UMMTO, Algeria) in 2014. Actually, she is a PhD student at LARI laboratory of UMMTO and XLIM laboratory (UMR CNRS 7252/Universit? de Limoges, France). Her research interests include Mobile and Wireless Networks, RFIDs Systems and Security.

Malika Belkadi received the PhD degree in computer science from Mouloud Mammeri University of Tizi Ouzou (UMMTO, Algeria). She is currently a research member at LARI laboratory and an Associate Professor at UMMTO. Her current research areas cover Mobile and Wireless Networks, RFID Systems, Security, IoT, Healthcare, Embedded Systems.



Olivier Blazy is an Associate Professor at the University of Limoges in France. He did his PhD in 2012 on proofs of knowledge and their application to blind signatures and authenticated key exchange. Since then, he leads the master level security program in Limoges, and conduct research on the areas of implicit cryptography, and identity-based communications. He is leading the ANR project: IDFix on ID-based communications.

Emmanuel Conchon received the MSc and PhD degrees in Wireless Communication from the "Institut National Polytechnique de Toulouse" (INPT), Toulouse, France in 2002 and 2006, respectively. In September 2008, he joined the Champollion University as an Associate Professor and IRIT (Institut de Recherche en Informatique de Toulouse) as a researcher. Since September 2015, he is an Associate Professor at the University of Limoges and researcher at XLIM. His research interests include security solutions for wireless networks, context-aware systems and secure middleware solutions for health applications.

Damien Sauveron received his MSc and PhD degrees in Computer Science from the University of Bordeaux, France. He has been Associate Professor with Habilitation at the XLIM Laboratory (UMR CNRS 7252, University of Limoges, France) since 2006. He is Head of the Computer Science Department in the Faculty of Science and Technology at the University of Limoges. Since 2011, he has been a member of the CNU 27, the National Council of Universities (for France). He has been chair of IFIP WG 11.2 Pervasive Systems Security since 2014, having previously been appointed vice-chair of the working group. His research interests are related to smart card applications and security (at hardware and software level), RFID/NFC applications and security, mobile network applications and security (particularly UAV), sensor network applications and security, Internet of Things (IoT) security, cyber-physical systems security, and security certification processes. In December 2013, the General Assembly of IFIP (International Federation for Information Processing) awarded Dr Sauveron the IFIP Silver Core award for his work. He has been involved in more than 100 research events in a range of capacities (including PC chair, General Chair, Publicity Chair, Editor/Guest Editor, Steering Committee member, and Program Committee member). He has served as external reviewer for several PhD thesis in foreign countries and in France. More on http://damien.sauveron.fr/.

Affiliations

Fatma Merabet[1,2] · Amina Cherif[1,2] · Malika Belkadi[1] · Olivier Blazy[2] · Emmanuel Conchon[2] · Damien Sauveron[2]

Fatma Merabet
merfatma@gmail.com

Amina Cherif
cherifamena@gmail.com

Malika Belkadi
belkadi_dz@yahoo.fr

Olivier Blazy
olivier.blazy@unilim.fr

Emmanuel Conchon
emmanuel.conchon@unilim.fr

1 Laboratoire LARI, Computer Science Department, Université de Tizi Ouzou, Tizi Ouzou, Algeria

2 MathIS, XLIM (UMR CNRS 7252/Université de Limoges), Limoges, France