Secret Sharing-based Authenticated Key Agreement Protocol

Petr Dzurenda
dzurenda@vutbr.cz
Brno University of Technology
Czech Republic

Sara Ricci
ricci@vutbr.cz
Brno University of Technology
Czech Republic

Raúl Casanova-Marqués
casanova@vutbr.cz
Brno University of Technology
Czech Republic

Jan Hajny
hajny@vutbr.cz
Brno University of Technology
Czech Republic

Petr Cika
cika@vutbr.cz
Brno University of Technology
Czech Republic

ABSTRACT

In this article, we present two novel authenticated key agreement (AKA) schemes that are easily implementable and efficient even on constrained devices. Both schemes are constructed over elliptic curves and extend Schonorr's signature of knowledge protocol. To the best of our knowledge, we introduce a first AKA protocol based on the proof of knowledge concept. This concept allows a client to prove its identity to a server via secret information while the server can learn nothing about the secret. Furthermore, we extend our protocol via secret sharing to support client multi-device authentication and multi-factor authentication features. In particular, the secret of the client can be distributed among the client's devices.

The experimental analysis shows that our secret sharing AKA (SSAKA) can establish a secure communication channel in less than 600 ms for one secondary device and 128-bit security strength. The protocol is fast even on very constrained secondary devices, where in most of cases takes less than 500 ms. Note that the time consumption depends on the computational capabilities of the hardware.

CCS CONCEPTS

• **Security and privacy** → **Multi-factor authentication**; Digital signatures; • **Theory of computation** → *Design and analysis of algorithms*; • **Computer systems organization** → Embedded hardware.

KEYWORDS

Authentication, Authenticated Key Agreement, Access Control, Cryptography, Proof of Knowledge, Security, Constrained Devices, Wearables, Internet of Things.

ACM Reference Format:
Petr Dzurenda, Sara Ricci, Raúl Casanova-Marqués, Jan Hajny, and Petr Cika. 2021. Secret Sharing-based Authenticated Key Agreement Protocol. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3465481.3470057

1 INTRODUCTION

Key agreement protocols are currently one of the most used cryptographic primitives. These schemes permit to set-up a secure communication channel between two or more entities. The first modern key agreement protocol was the Diffie-Hellman (DH) protocol [8]. However, the basic DH protocol suffers from the man-in-the-middle attack since the involved parts do not authenticate themselves. The combination of a key agreement protocol with a digital signature scheme allows obtaining an AKA protocol. AKA protocols not only allow parties to compute the session key but also ensure the authenticity of the involved parties. AKA protocol is the process by which two (or more) authenticated entities establish a shared secret key. The key is then used to achieve confidentiality or data integrity, for instance.

AKA protocols find applications mainly in Internet of Things (IoT) environment. There are many applications based on IoT such as home automation, smart city, healthcare, smart grid, and smart car [16]. These applications generate a large amount of data which leads to many security concerns, such as data leakages, eavesdropping, or unauthorized access. Unfortunately, IoT devices are constrained in their computational and memory capabilities. Therefore, we need a lightweight authentication protocol to cope with such an IoT environment.

This work presents an authenticated key agreement protocol that is provable secure, easily implementable, and efficient even on constrained devices in IoT. Furthermore, our protocol supports user multi-device authentication and it is based on zero-knowledge proofs and elliptic curve constructions.

1.1 Related work

Blake and Menezes [5] give a good overview of key agreement protocols that are based on the intractability of the DH problem. Law et al. [12] propose a two-pass protocol for authenticated DH key agreement which works on elliptic curves. This scheme combines static and ephemeral key pairs to obtain the session key.

Several identity-based key DH AKA are based on Weil and Tate pairing [7, 14, 24, 26, 27]. However, all these schemes are proved to be insecure [28]. To be noted that Smart [27] developed an identity-based key AKA which requires a trusted key generation center and uses a secret sharing scheme for the key generation. Wang [28] presents a pairing-based identity-based AKA which achieves all security AKA properties.

Pasini and Vaudenay [19] propose three protocols. The first scheme uses DH protocol in a secure channel. The second scheme combines the DH protocol and a short authenticated strings. The short string is used for authentication on an insecure channel. Finally, the third scheme uses an universal hash function. All schemes are based on commitment scheme. They focus on minimizing the number of message moves over the insecure channel and the length of authenticated messages.

Reddy et al. [22] extensively studied the existing mutual authentication protocols for a multi-server environment. Moreover, they propose a biometric-based 3-factor mutually AKA protocols for multi-server architecture based on elliptic curve cryptography and Burrows–Abadi–Needham logic. In 2019, the same authors [21] present an anonymous three-factor mutually AKA protocol for client–server architecture on elliptic curve cryptography. Yang et al. [29] present an AKA protocol with dynamic credential for wireless sensor networks. Dynamic credentials allow to clients to know if their authentication credentials have been compromised.

Melki et al. [15] propose a lightweight and multi-factor authentication protocol for IoT devices. The protocol is based on configurable Physical Unclonable Functions (PUF) and channel-based parameters. Hajny et. al. [9] present a multi-device authentication scheme using wearables and IoT devices. This scheme is provable secure and based on zero-knowledge proofs. However, the scheme does not support mutual authentication and key agreement mechanisms. In 2018, the same authors [10] present multi-device authentication scheme with strong privacy protection.

The scheme is provably secure and provides the full set of privacy-enhancing features such as the anonymity, untraceability, and unlinkability of users. Even this scheme does not provide mutual authentication and key agreement mechanisms. The scheme is symmetric, therefore, it does not provide non-repudiation property so typical for digital signatures. Lopes et al. [13] propose a secret-sharing-based AKA based on Shamir secret share scheme [25] for multi-device authentication. In their scheme, a trusted third party create and share the secret keys among the user's devices. Moreover, each partial secret key needs to be disclose to be authenticated.

To the best of our knowledge, we introduce a first AKA protocol based on the proof of knowledge concept. This concept allows the user to prove its identity to a server via secret information (such as a password or cryptographic key) while the server can learn nothing about this secret. Furthermore, our AKA is based on Schnorr signature [23] which produces fast and short signatures. Finally, the protocol construction easily integrates other security and privacy features such as multi-device and multi-factor authentication techniques.

1.2 Contributions

In this article, we present two novel schemes, namely AKA and SSAKA. The basic AKA scheme is based on zero-knowledge-proof protocols and it is efficient even on constrained devices that are very often used in current IoT ecosystems. To the best of our knowledge, we introduce a first AKA protocol based on the proof of knowledge concept. The AKA zero-knowledge core allows to increase the security strength of the algorithm by sharing the client's secret among more user devices such as wearable, embedded microcontrollers, or smartcards. Moreover, the client can actively be part of the authentication by using passwords or PIN codes. Thanks to this, our AKA can be easily extended to support multi-device and multi-factor authentication features. This is achieved in SSAKA scheme where our AKA scheme is combined with a slightly modified multi-device authentication technique presented in [9]. The SSAKA strengthens security by sharing the client secret between more parties. In particular, the secret of the client can be distributed among the client devices. Finally, an evaluation of our schemes is presented. We show that our SSAKA can establish a secure communication channel in less than 600 ms for one secondary device and 128-bit security strength.

This paper is organized as follows: Section 2 defines used notation and gives the necessary background on underlying cryptographic primitives used in our AKA schemes. Section 3 presents our basic AKA scheme based on zero-knowledge proofs and Section 4 introduces our secret sharing-based AKA which allows deployment of more additional devices and user password to the authentication and key agreement processes. Section 5 discusses security analysis of our schemes and Section 6 shows our implementation results. In the last section, we conclude this work.

2 PRELIMINARIES

In this section, we introduce the notation used throughout the paper. Moreover, we recall proof of knowledge concept and secret sharing scheme which are the cryptographic core of our AKA protocol. In particular, we will focus on signature of knowledge concepts which are non-interactive proof of knowledge protocols.

From now on, the symbol ":" means "such that", "$|x|$" is the bitlength of x, and "$||$" denotes the concatenation of two binary strings. We write $a \xleftarrow{\$} A$ when a is sampled uniformly at random from A. A secure hash function is denoted as $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^\kappa$, where κ is a security parameter. Moreover, $\mathbb{G} = \langle g \rangle$ denotes a cyclic group of order q, and α is an element of $\mathbb{Z}_q^*$. We describe the Proof of Knowledge (PK) and the Signature of Knowledge (SK) protocols using the notation introduced by Camenisch and Stadler. In particular, the protocol for proving the knowledge of discrete logarithm of c with respect to g is denoted as $\mathsf{PK}\{\alpha : c = g^\alpha\}$ and the protocol for proving the knowledge of discrete logarithm of c with respect to g and message m is denoted as $\mathsf{SK}\{\alpha : c = g^\alpha\}(m)$.

2.1 Proof of Knowledge

The concept of proof of knowledge is frequently used in many modern cryptosystems such as group signatures, ring signatures, and attribute-based credentials. In proof of knowledge, one entity, namely prover, proves to another entity, namely verifier, the veracity of a given statement. Statements regarding discrete logarithm problems can be easily proven by using Σ-protocols [6].

A Σ-protocol is a simple 3-way protocol where the prover commits a random number r, receives a challenge e, and finally responds by the proof z to the challenge. One of the most used Σ-protocol is the Schnorr protocol [6]. We recall its properties below:

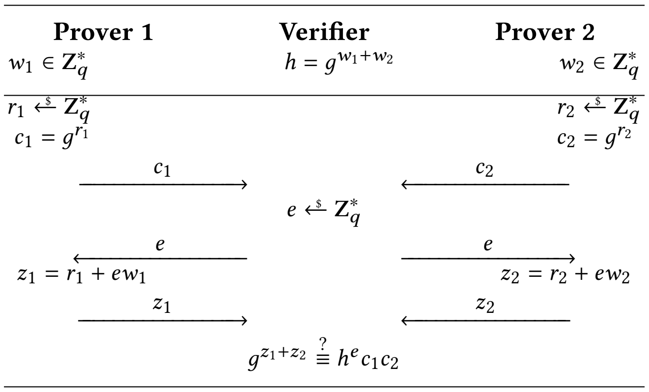

Figure 1: Proof of knowledge of the sum of discrete logarithms PK$\{w_1 + w_2 : h = g^{w_1+w_2}\}$.

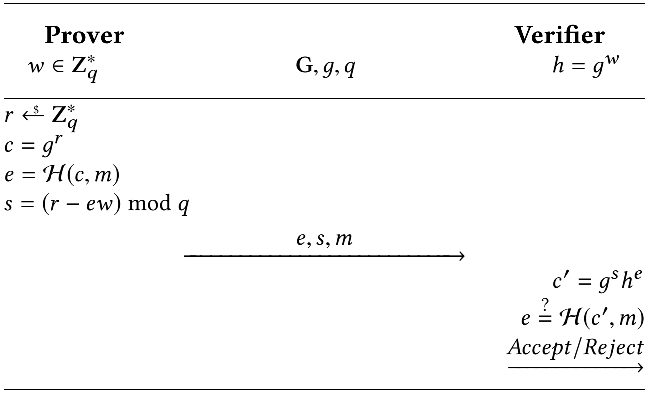

Figure 2: Schnorr's signature of knowledge of discrete logarithm SK$\{w : h = g^w\}(m)$.

- `Completeness`: if the statement is true, the verifier will always accept the proof.
- `Special Soundness`: if the statement is false, no cheating prover can convince the verifier of the validity of the proof, and the prover will be always rejected.
- `Special Honest Verifier Zero-Knowledge`: if the statement is true, no verifier learns anything other than the fact that the statement is true.

Okamoto Σ-protocol [18] is an extension of Schnorr protocol which allows the usage of two private keys and provides witness indistinguishability. The Okamoto protocol is defined as follow: Let $\mathbb{G}$ be a group of prime order q and $g_1, g_2 \in \langle g \rangle$ are generators such that the $log_{g_1} g_2$ is known to nobody. Let $w_1, w_2 \in \mathbb{Z}_q^*$ be the prover's private keys, and let $h = g_1^{w_1} g_2^{w_2}$ be the prover's public key. The Okamoto protocol is proof of knowledge of discrete logarithm of c with respect to g_1, g_2 donated as PK$\{w_1, w_2 : h = g_1^{w_1} g_2^{w_2}\}$. In our protocol, we adopt the Okamoto protocol into one single group generator in order to design secret sharing of knowledge protocols, see Figure 1.

Interactive proof of knowledge protocols are frequently used in authentication schemes, where a challenge e is generated by a verifier. On the other hand, non-interactive proof of knowledge protocols are widely used in particular for signature scheme constructions. In this case, the challenge e is generated by the prover with the use of a secure hash function $\mathcal{H}$. The non-interactive variant is more often called signature of knowledge, due to the inclusion of the message in the proof as shown in Figure 2.

2.2 Secret Sharing Scheme

Secret Sharing [4] is a cryptographic tool that is used as a building block in many protocols such as multiparty computation, generalized oblivious transfer, and attribute-based encryption. The scheme involves a dealer who owns a secret and a set of n parties. The dealer distributes a secret value k in shares in a way that only qualified subsets of parties can recover the secret.

A secret sharing scheme has two requirements [4]:

- `Correctness`: the secret k can be reconstructed by any authorized set of parties.
- `Perfect Privacy`: every unauthorized set cannot learn anything about the secret (in the information theoretic sense) from their shares.

The minimum number of shares needed to reconstruct the secret is called threshold. There are constructions with and without threshold [11, 25].

3 BASIC AKA PROTOCOL

In this section, we define the algorithms and entities of our AKA protocol. The AKA communication pattern is depicted in Figure 3 and employs two parties:

- `Client`: is typically a user which is accessing specific online services. To get an access to these services, the client must pass the authentication phase by proving possession of the access key. The key is stored on a user device which can be any computing device represented by simple microcontrollers, wearables, smartphones, or even by powerful personal computers.
- `Server`: is very often a powerful computer providing different user services. However, the rise of IoT environments causes that servers are represented more often by computationally less powerful devices such as Raspberry Pi single-board computers.

3.1 Protocol description

The goal of the AKA protocol is not only to allow parties to compute the session key but also to ensure the authenticity of both involved parties. To do so, the client holds the servers' signature public key pk_S and the server holds the client's signature public key pk_S. On both sides, we use a variant of Schnorr's signature scheme to generate the signatures and, therefore, to authenticate both communication parties. We assume, that both these signature public keys are securely transferred between parties before AKA protocol starts. In the case of the server, it can be easily achieved by using a Public Key Infrastructure (PKI), which involves trusted

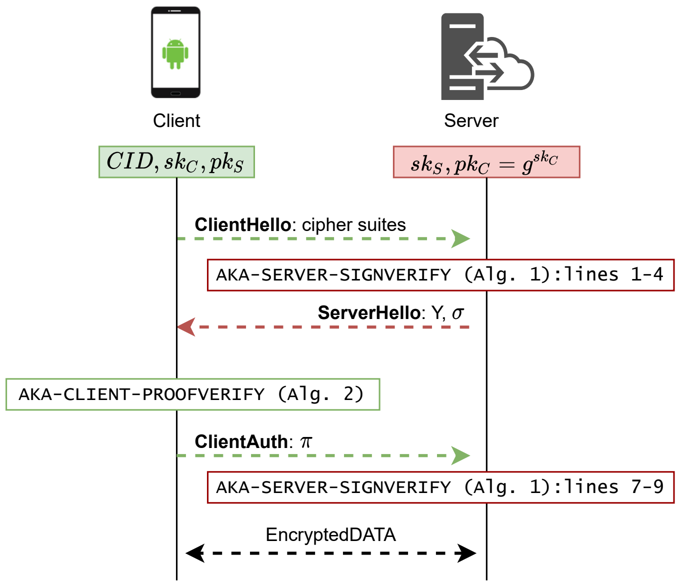
Client
Server
CID, sk_C, pk_S
$sk_S, pk_C = g^{sk_C}$
ClientHello: cipher suites
AKA-SERVER-SIGNVERIFY (Alg. 1):lines 1-4
ServerHello: Y, σ
AKA-CLIENT-PROOFVERIFY (Alg. 2)
ClientAuth: π
AKA-SERVER-SIGNVERIFY (Alg. 1):lines 7-9
EncryptedDATA

Figure 3: Basic AKA protocol.

certification authorities, or by pre-installing the server's signature public keys directly to the client application. The client usually provides his/her signature key during the registration phase. Then the AKA protocol works in three following steps:

(1) `Step 1`: The client sends `ClientHello` message with 1) his unique Client IDentifier (CID), 2) the cipher suites specifying preferred cryptographic algorithms (e.g., elliptic curve type such as SECP256), symmetric cipher (eg., AES-GCM-256), and hash function (e.g., SHA2-256), and 3) the client challenge RNDc. The RNDc challenge protects the system against authentication replay attacks.

(2) `Step 2`: The server creates a message `Y` which includes CID, used cipher suites, RNDc, and a confirmation of common cipher suites. The message can have for example the following format: `Y = {CID || SECP256 || SHA2-256 || AES128GCM || RNDc || ACCEPT}`. The server then executes `AKA-SERVER-SIGNVERIFY` algorithm to generate the Schnorr' signature σ on the message `Y` and, therefore, to generate authentication proof. Both values `Y` and σ are sent back to the client in the `ServerHello` message. It is important to notice that the signature includes also the server's DH public key for the key agreement protocol. This DH key serves also as an authentication challenge of the server since the key is randomly generated for each communication session. The client then performs `AKA-CLIENT-PROOFVERIFY` algorithm to check the validity of the signature σ, computes a common session key κ, and client's proof of knowledge π of the secret key sk_C.

(3) `Step 3`: The server performs second part of the `AKA-SERVER-SIGNVERIFY` algorithm to compute the common session key κ and verify client's authentication proof.

Thanks to the deployment of the Schnorr's signature, we can 1) sign messages of both parties and, therefore, authenticate one against other, 2) generate randomness, i.e. an authentication challenge which allows avoiding replay attacks, and 3) perform DH key agreement protocol over cryptographic commitments to establish a common session key.

3.2 Definition of algorithms

In this section, we present the concrete instantiations of cryptographic algorithms of our AKA.

$(spar) \leftarrow$ `AKA-SETUP`(κ):
The protocol is run by a server in order to generate initial system parameters. The algorithm inputs the security parameter κ and outputs the system parameters *spar*. On the input of the security parameter κ the server generates a cyclic group $\mathbf{G} = \langle g \rangle$ of prime order $q : |q| = \kappa$, where the Discrete Logarithm (DL) assumption holds and outputs $spar = (\mathbf{G}, g, q)$ as public system parameters. The systems parameters are securely distributed between all system parties and they are used as an implicit input for all following algorithms.

$(sk_C, pk_C) \leftarrow$ `AKA-CLIENT-REGISTER`(κ):
The protocol is run between a server and a client. At first, the client establishes a secure communication channel with the server. At second, the algorithm inputs the security parameter κ and outputs user key pairs $sk_C \xleftarrow{\$} \mathbb{Z}_q^*, pk_C = g^{sk_C}$. The secret key sk_C is securely stored on the client's device, while the public key pk_C together with the user login name are sent to the server. The server assigns the unique CID to the client. CID, login name, and the public key pk_C are securely stored in the server database.

$(\tau_S, \kappa) \leftarrow$ `AKA-SERVER-SIGNVERIFY`(Y, sk_S, pk_C):
The algorithm takes as input a message `Y`, the server's secret key sk_S, and the client's signature public key pk_C. It outputs the result of client authentication $\tau_S = 0/1$, and the shared session key κ as shown in Algorithm 1. At first, the server computes a cryptographic commitment $t_S = g^{r_S}$ as a part of the Schnorr's signature. Furthermore, the commitment serves also as an authentication challenge of the server and DH public key of the server, while the value r_S represents DH secret key. Both `Y` and σ are sent to the client, while the client answers with his/her authentication proof π. The server verifies the reconstruct of the client's cryptographic commitment $t' = g^{s_C} \cdot pk_C^{e_C}$. The commitment t' also represents the client's DH public key, and therefore, the client computes a common session key such as $\kappa = t'^{r_S}$. Finally the server very the client's proof $e_C \stackrel{?}{=} \mathcal{H}(Y, t', \kappa)$.
$(\tau_C, \pi, \kappa) \leftarrow$ `AKA-CLIENT-PROOFVERIFY`(Y, σ, pk_S, sk_C):
The algorithm takes as input a message `Y`, the server's signature public key pk_S, and the client's secret key sk_C. It outputs the result of the server authentication $\tau_C = 0/1$, the client authentication proof π, and the common shared session key κ as shown in Algorithm 2. At first, the client reconstructs the server's commitment $t'_S = g^{s_S} \cdot pk_S^{e_S}$, which also represents the server's DH public key and checks it's correctness by comparing $e_S \stackrel{?}{=} \mathcal{H}(Y, t'_S)$. At second, if the signature is valid, the client computes the authentication

Algorithm 1 AKA-SERVER-SIGNVERIFY(Y, sk_S, pk_C)

```
1: $r_S \xleftarrow{\$} \mathbb{Z}_q^*$
2: $t_S \leftarrow g^{r_S}$
3: $e_S \leftarrow \mathcal{H}(Y, t_S)$
4: $s_S \leftarrow r_S - e_S \cdot sk_s \bmod q$

5: Send: $Y, \sigma = (e_S, s_S)$          ▷ AKA-CLIENT-PROOFVERIFY
6: Receive: $\pi = (e_C, s_C)$            ◁ AKA-CLIENT-PROOFVERIFY
7: $t' \leftarrow g^{s_C} \cdot pk_C^{e_C}$
8: $\kappa \leftarrow t'^{r_S}$
9: $\tau_S \leftarrow e_C \stackrel{?}{=} \mathcal{H}(Y, t', \kappa)$
10: return $\tau_S = 0/1, \kappa$
```

proof π and a common DH session key $\kappa = t_S'^{r_C}$. Otherwise, the client ends the protocol.

Algorithm 2 AKA-CLIENT-PROOFVERIFY(Y, σ, pk_S, sk_C)

```
1: $t_S' \leftarrow g^{s_S} \cdot pk_S^{e_S}$
2: $\tau_C \leftarrow e_S \stackrel{?}{=} \mathcal{H}(Y, t_S')$
3: if $\tau_C = 0$ then exit
4: $r_C \xleftarrow{\$} \mathbb{Z}_q^*$
5: $t \leftarrow g^{r_C}$
6: $\kappa \leftarrow t_S'^{r_C}$
7: $e_C \leftarrow \mathcal{H}(Y, t, \kappa)$
8: $s_C \leftarrow r_C - e_C \cdot sk_C \bmod q$
9: return $\tau_C = 0/1, \pi = (e_C, s_C), \kappa$
```

4 SECRET SHARING-BASED AKA PROTOCOL

Our Secret Sharing-based Authenticated Key Agreement (SSAKA) protocol is built upon our AKA protocol which was presented in Section 3. The SSAKA extends AKA scheme by secret sharing property. The secret is reconstructed only if all the involved parties (devices) collaborate. The client decides the number of involved devices and if being actively part of the protocol by using a password or a PIN code, for instance.

In this section, we define the algorithms and protocols of our SSAKA scheme in more details. The communication pattern is depicted in Figure 4 and employs four types of entities:

- Server: is usually a service provider that needs to verify the identity of their users. The server can be represented by a powerful computer or a less powerful single board computer.
- Client: is represented by the user's Master Device (MD) which can be a PC, laptop, smartphone, or tablet. The client needs to prove his/her identity and to establish a secure communication channel with the server.
- Device: is an additional Secondary Device (SD) with usually less computational power. Examples of SD devices are smartcards, smartwatches, sensors, and embedded microcontrollers. SDs are involved in the authentication process to strengthen security.
- Party (P): specifies a device, a password or a PIN code used in the protocol.

SSAKA protocol requires the addition of three new algorithms which allow extending our AKA protocol to work in a multi-device environment. In particular, SSAKA-CLIENT-ADDSHARE algorithm generates a new client's secret key, where each involved device knows only its share of the key. SSAKA-CLIENT-REVSHARE algorithm allows revoking a device secret key from the client key. At last, SSAKA-DEVICE-PROOF algorithm computes the proof of knowledge of each device's secret. Moreover, Algorithm 2 is slightly modified to Algorithm 3. These changes permit the protocol to run in a multi-device environment. The sub-protocols are specified below in details.

$(\bar{pk_C}, pk_C, sk_C) \leftarrow$ SSAKA-CLIENT-ADDSHARE($\langle sk_i \rangle_{\in NEW}, sk_C, pk_C$): The protocol is run between a client MD, several SDs and the server. The task of this algorithm is to spread the client secret sk_C among all involved parties. The algorithm inputs system parameters *spar*, client secret key $sk_C = sk_0 + \sum_{i=1}^{DEV} sk_i$, where sk_0 is the MD share and $sk_1, \ldots sk_{DEV}$ are shares of DEV already registered SDs, and $\langle sk_i \rangle_{\in NEW}$ represents the set of new shares (i.e., newly registered SDs). In this case, each SD generates its own keypair $sk_i \xleftarrow{\$} \mathbb{Z}_q^*$, $pk_i = g^{sk_i}$. The secret keys are kept secret while the public keys are securely sent to MD. The piece of secret can be also shared with the client him/herself. In this case, the client share sk_{pw} is computed by hashing his/her password, or PIN code. Finally, the client MD computes the extension of the client public key, i.e it computes $\bar{pk_C} = \prod_{i=1}^{NEW} pk_i$ for all NEW available devices or for the eventually client's passwords or PIN codes. The client runs the SSAKA protocol and sends the extension of the public key $\bar{pk_C}$ to the server through established secure channel. The server updates the main client's public key $pk_C = pk_C \cdot \bar{pk_C}$. After this update, the new devices, passwords, or PIN codes must be always used in the SSAKA protocol. Let note that the MD private key sk_0, and SDs keys $\langle sk_i \rangle_{\in DEV}$ never leave the devices, only the public keys pk_C $\langle pk_i, \rangle_{\in DEV}$ are revealed. Each party (i.e., client's device) knows only a share of the client secret sk_C and only together they are able to forge it, that is $sk_C = sk_0 + \sum_{i=1}^{DEV} sk_i$, where DEV is the number of all registered SDs.

$(\bar{pk_C}, pk_C, sk_C) \leftarrow$ SSAKA-CLIENT-REVSHARE($\langle sk_i \rangle_{\in REV}, sk_C, pk_C$): The protocol is run between a client MD, several SDs and the server. The task of the algorithm is to revoke the client secret among the parties. The algorithm inputs system parameters *spar*, client secret key $sk_C = sk_0 + \sum_{i=1}^{DEV} sk_i$, where sk_0 is the MD share and $sk_1, \ldots sk_{DEV}$ are shares of DEV already registered SDs, and $\langle sk_i \rangle_{\in REV}$ represents the set of revoked shares (i.e., newly revoked SDs). The client selects the shares to be revoked from the client secret sk_C. To do so, the clients picks all corresponding SDs public keys or eventually public key of the user password $pk_{pw} = g^{sk_{pw}}$, where $sk_{pw} = \mathcal{H}(password)$. Finally, the client's MD computes the subtraction of client's public key, i.e it computes $\bar{pk_C} = \prod_{i=1}^{REV} pk_i$ for all REV devices or eventually clients passwords or PIN codes. The MD runs SSAKA protocol and sends to server the subtraction of the public keys $\bar{pk_C}$ through established secure channel. The server then updates the main client's public key $pk_C = pk_C \cdot \bar{pk_C}^{-1}$. After

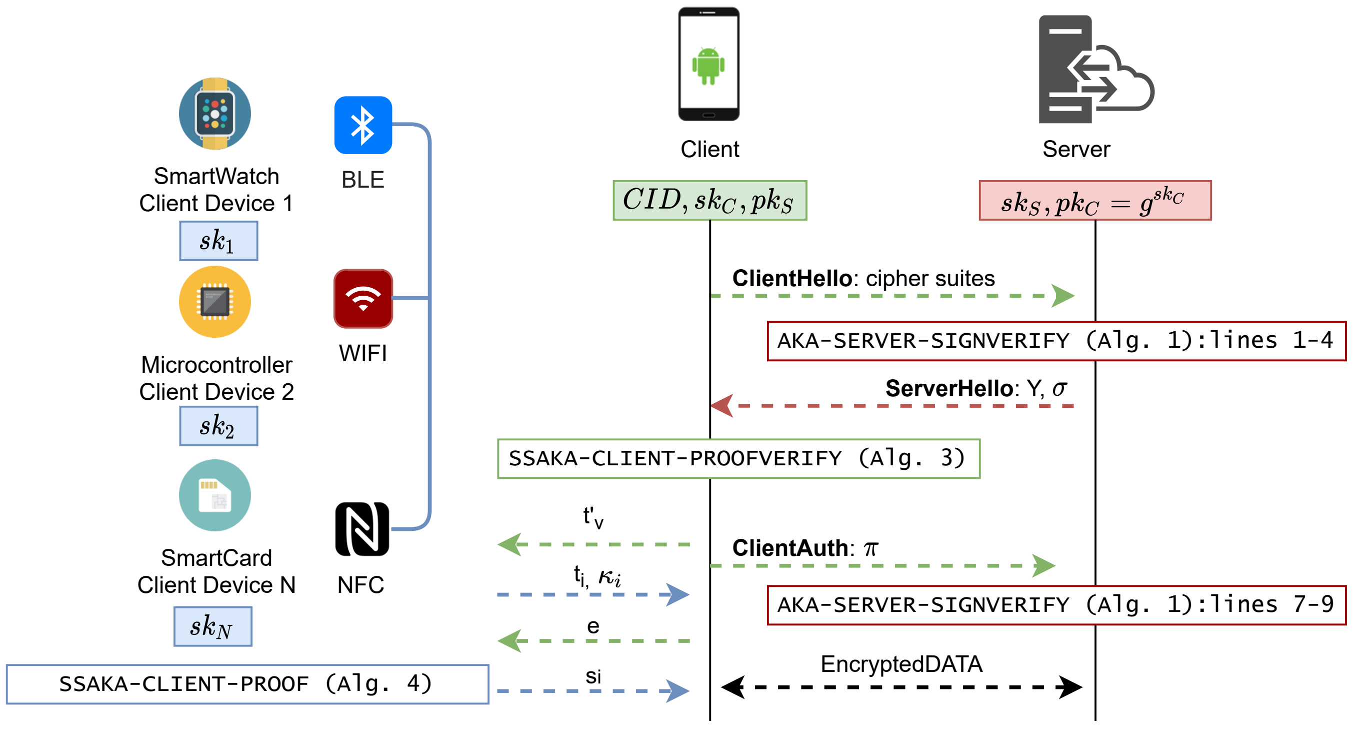
SmartWatch Client Device 1
sk_1
Microcontroller Client Device 2
sk_2
SmartCard Client Device N
sk_N
BLE
WIFI
NFC
Client
Server
CID, sk_C, pk_S
$sk_S, pk_C = g^{sk_C}$
ClientHello: cipher suites
AKA-SERVER-SIGNVERIFY (Alg. 1):lines 1-4
ServerHello: Y, σ
SSAKA-CLIENT-PROOFVERIFY (Alg. 3)
t'v
ti, κ_i
e
si
ClientAuth: π
AKA-SERVER-SIGNVERIFY (Alg. 1):lines 7-9
EncryptedDATA
SSAKA-CLIENT-PROOF (Alg. 4)

Figure 4: SSAKA protocol using multi-device access control.

Algorithm 3 SSAKA-CLIENT-PROOFVERIFY(Y, σ, pk_S, sk_C)

```
1:  $t'_S \leftarrow g^{s_S} \cdot pk_S^{e_S}$
2:  $\tau_C \leftarrow e_S \overset{?}{=} \mathcal{H}(Y, t'_S)$
3:  if $\tau_C = 0$ than exit

4:  Send: $t'_S$                                  ▷ SSAKA-DEVICE-PROOF
5:  Receive: $\langle\kappa_i\rangle_{\in DEV}, \langle t_i\rangle_{\in DEV}$     ◁ SSAKA-DEVICE-PROOF

6:  $r_0 \xleftarrow{\$} \mathbb{Z}_q^*$
7:  $t \leftarrow g^{r_0} \cdot \prod_{i=1}^{DEV} t_i$
8:  $\kappa \leftarrow t_S'^{r_C} \cdot \prod_{i=1}^{DEV} \kappa_i$
9:  $e_C \leftarrow \mathcal{H}(Y, t, \kappa)$

10: Send: $e_C$                                   ▷ SSAKA-DEVICE-PROOF
11: Receive: $\langle s_i\rangle_{\in DEV}$                     ◁ SSAKA-DEVICE-PROOF

12: $s_0 \leftarrow r_0 - e_C \cdot sk_0 \mod q$
13: $s_C \leftarrow \sum_{i=0}^{DEV} sk_i \mod q$
14: Return $\tau_C = 0/1, \pi = (e_C, s_C), \kappa$
```

this update, the revoked Ps are no more used in the SSAKA protocol.

(τ_C, π, κ) $\leftarrow$ SSAKA-CLIENT-PROOFVERIFY(Y, σ, pk_S, sk_C):
The algorithm runs as Algorithm 2 with small modifications that are marked in red in Algorithm 3. In this case, the algorithm requires to call Algorithm 4 for each SD, which has a share of the secret, and therefore is involved in the protocol. Moreover, t and κ values need the commitments t_i and κ_i of each SD. Values s_i represent proofs of knowledge of the secret shares of all SDs.

Algorithm 4 SSAKA-DEVICE-PROOF(t'_S, sk_i)

```
1: $r_i \xleftarrow{\$} \mathbb{Z}_q^*$
2: $t_i \leftarrow g^{r_i}$
3: $\kappa_i \leftarrow t_S'^{r_i}$

4: Send: $\kappa_i, t_i$                    ▷ SSAKA-CLIENT-PROOFVERIFY
5: Receive: $e_C$                        ◁ SSAKA-CLIENT-PROOFVERIFY

6: $s_i \leftarrow r_i - e_C \cdot sk_i \mod q$
7: Return: $s_i, \kappa_i$
```

(s_i, κ_i) $\leftarrow$ SSAKA-DEVICE-PROOF(t'_S, sk_i):
The algorithm takes as input server's DH public key t'_S and the P's secret key sk_i and outputs the P's authentication proof s_i and P's DH public key fragment κ_i, see Algorithm 4 for more details. At first, the device commits to random value $t_i = g^{r_i}$ and computes DH public key fragment as $\kappa_i = t_S'^{r_i}$. Both values sends to MD. The MD responses with authentication challenge e_C on which the device computes the proof of knowledge s_i its share of client secret sk_i.

5 SECURITY ANALYSES

We prove the security of our AKA and SSAKA protocols in this section. The AKA protocol is based on provable secure cryptographic primitives, namely Schnorr signature and DH protocol.

Lemma 5.1. *The Schnorr signature is existentially unforgeable under chosen-message attacks in the random oracle model assuming that the DL problem is hard.*

Proof. The Lemma 5.1 is proven in [20]. □

Lemma 5.2. *The AKA and SSAKA authentication protocols are both complete.*

Proof. We consider the verification equation of Algorithm 1 The server verifies the equation $e_C \overset{?}{=} \mathcal{H}(Y, t', \kappa)$, where the $\kappa = t'^{r_S}$ and t' is the client's commitment. Therefore, the protocol pass if the commitment is correctly reconstructed.

$$t' = g^{s_C} \cdot pk_C^{e_C} = g^{r_C - e_C \cdot sk_C} \cdot g^{sk_C \cdot e_C} = g^{r_C} = t$$

In case of SSAKA, the situation is equivalent, since we can write $sk_C = sk_0 + \sum_{i=1}^{DEV} sk_i$, $r_C = r_0 + \sum_{i=1}^{DEV} r_i$, and $pk_C = g^{sk_0 + \sum_{i=1}^{DEV} sk_i}$), where DEV is the number of all registered SDs.

□

Lemma 5.3. *The AKA and SSAKA authentication protocols are both sound.*

Proof. Suppose that a client does not know the private key and is ready to correctly respond to at least two challenges (denoted as e_C, e'_C) by sending s_C and s'_C. Then, the following equations must hold for the client to be accepted.

$$t' = g^{s_C} \cdot pk_C^{e_C}$$

$$t' = g^{s'_C} \cdot pk_C^{e'_C}$$

By dividing we get:

$$1 = pk_C^{e_C - e'_C} \cdot g^{s_C - s'_C}$$

And finally we get:

$$pk_C = g^{\frac{s'_C - s_C}{e_C - e'_C}}$$

And we reached the contradiction because the user knows the private key $sk_C = \frac{s'_C - s_C}{e_C - e'_C}$.

SSAKA soundness proof directly follows from AKA one. In fact, the only difference is in the values sk_C and pk_C which do not change the proof as shown in Lemma 5.2

□

Lemma 5.4. *The AKA and SSAKA authentication protocols are both zero-knowledge.*

Proof. We prove the zero-knowledge property by constructing the zero-knowledge simulator S. The simulator works in the following steps.

(1) Randomly selects the response $\hat{s_C} \xleftarrow{\$} \mathbb{Z}_q^*$.
(2) Randomly selects the challenge $\hat{e_C} \xleftarrow{\$} \mathbb{Z}_q^*$.
(3) Computes the commitment $\hat{t} = pk_C^{\hat{e_C}} \cdot g^{\hat{s_C}}$.

The simulator's output is computationally indistinguishable from the real protocol transcript, i.e. $(\hat{t}, \hat{e_C}, \hat{s_C},) \cong_c (t, e_C, s_C)$, because all pairs are selected randomly and uniformly from the same sets.

SSAKA zero-knowledge proof directly follows from AKA one as in Lemmas 5.2 and 5.3.

□

Lemma 5.5. *AKA and SSAKA key agreement protocol are secure against eavesdroppers.*

Proof. The AKA key agreement protocol is based on DH protocol, where both entities reconstruct the DH public key of the counterparty from the authentication phase. In particular, the entities reconstruct cryptographic commitments t'_S and t', and compute a common secret κ.

$$\kappa = t'^{r_C}_S = (g^{r_S})^{r_C} = (g^{r_C})^{r_S} = t'^{r_S}$$

In case of SSAKA, the situation is equivalent, since we can write $r_C = r_0 + \sum_{i=1}^{DEV} r_i$ and therefore, $t'_C = g^{r_0 + \sum_{i=1}^{DEV} r_i}$), where DEV is the number of all registered SDs.

□

Lemma 5.6. *SSAKA protocol provides both secret sharing properties, i.e. correctness and perfect privacy.*

Proof. The proof is straightforward.

- `Correctness`: since each party P_i owns a different part of the secret key sk_i, the user secret key $sk_C = \sum_i sk_i$ can be reconstructed only knowing all the sk_i.
- `Perfect privacy`: unauthorized set cannot learn anything about the secret since $sk_C = \sum_i sk_i \mod q$. The modulus prevent to have information on sk_C.

□

Furthermore, our AKA and SSAKA protocols meet all security attributes of authenticated key agreement protocol defined in [16]:

- `Mutual authentication`: both entities, i.e. a client and a server, authenticate each other during the run of the protocol. Our AKA uses the authentication model challenge-response based on the Schnorr's digital signature.
- `Key compromise impersonation`: if an adversary compromises the long-term secret key of one entity, then the adversary can impersonate only this entity no others system entities. Our AKA uses asymmetric cryptography, where each involved entity uses its own secret Schnorr key.
- `Parallel session attack`: the knowledge of some previous session keys does not allow the adversary to compromise other session keys. In both protocols session keys values are randomized, therefore knowing previous session keys does not leak any information on user's secret key (proof-of-knowledge property).
- `Denial of service attack`: an adversary can try to consume resources of the server by sending fake login messages. If that happens, the user's CID or IP address can be easily putted on the blacklist and the login request will be denied.
- `Replay attack`: our AKA and SSAKA use authentication model challenge-response which prevents against replay attacks.

- `Man-in-the-middle attack`: our AKA uses mutual authentication based on Schnorr signatures which prevents against man-in-the-middle attacks.

6 EXPERIMENTAL RESULTS

In this section, we show the evaluation of our AKA and SSAKA protocols. AKA protocol is efficient and easy to implement even on very constrained devices. Moreover, we implement the whole SSAKA protocol on a set of various modern programmable devices, namely smartwatches, smartcards, smartphones, microcontrollers, and microcomputers. The hardware (HW) and software (SW) specification of deployed devices is shown in Table 1.

We suggest using only the cryptographic algorithms recommended by international cybersecurity authorities such as the National Institute of Standards and Technology (NIST) and the European Union Agency for Cybersecurity (ENISA), see Table 2 for more details.

The implementation is based on RIOT [1] operating system, RIOT compatible devices, and libraries. It allows us to make the applications easily portable across various types of devices based on different architectures and developed by different manufactures. In particular, we build up the applications on `Crypto` and `micro-ecc` libraries written in C programming language:

- `Crypto` [1]: is the native cryptographic library of RIOT. It provides block ciphers, operation modes, and cryptographic hash algorithms. We consider it mainly for performing Advanced Encryption Standard (AES) encryption in the CCM operation mode and hashing with Secure Hash Algorithm (SHA) between a user device and the server.
- `micro-ecc` [2]: is a small and fast Elliptic Curve Diffie Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) implementation for 8-bit, 32-bit, and 64-bit processors. The library is RIOT compatible and supports secp160r1, secp192r1, secp224r1, secp256r1, and secp256k1 curves.

The `micro-ecc` library implements ECDH and ECDSA algorithms and provides Application Programming Interface (API) to use them. The library provides also an access to AKA required underlayer mathematical operations such as Elliptic Curve (EC) point scalar multiplication, modular reduction, multiplication, and addition. To make them available to developers, only the flag `-DuECC_ENABLE_VLI_API` needs to be added to the compiler. The only missing AKA required operation is EC points addition. We extended the `micro-ecc` library by the EC points addition operation (the function `uECC_point_add`). To do so, we slightly modify the `uECC_verify` function in the `uECC.c` file. This function is originally used in ECDSA verification protocol to compute $sum = G + Q$, where G, Q are two EC points.

The RIOT supports two main elliptic curve cryptographic libraries, namely `micro-ecc` and Relic [3]. Relic supports among others calculation on NIST $E(F_{2^m})$ and $E(F_p)$ curves, pairing-friendly curves, pairings and related extension fields F_{p^k}. Furthermore, the Relic shows significantly better performance results as shown in Figure 5. However, the Relic memory requirements are significantly higher than `micro-ecc` ones, and therefore, we avoid the Relic from the AKA protocol implementations. We consider the `Crypto` library to implement ciphers algorithms since it is part of the RIOT and due to its efficiency. We compared the `Crypto` with TinyCrypt library, where the TinyCrypt was ca. 6 times slower. The Figure 5 shows the time needed to encrypt 16 KB data with AES256-CCM for `Crypto` library and to multiply the EC point with the scalar (ecMUL) with Relic and `micro-ecc` libraries.

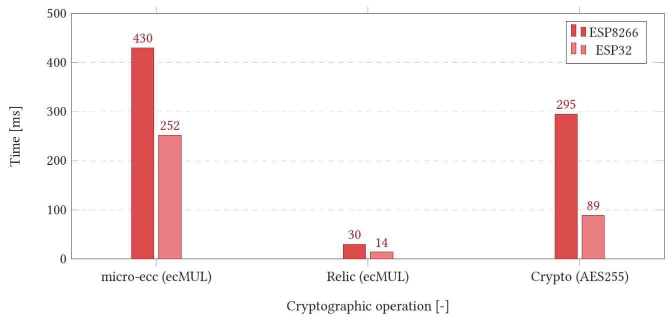

Figure 5: Efficiency of C libraries on different MCU units.

We use `micro-ecc` library also for Android implementations in smartphones and smartwatches. The Android Native Development Kit (NDK) allows us to execute a program in C/C++ on Android devices instead of using Java libraries. In fact, we compare the efficiency of Spongy Castle [17] (Android version of Java Bouncy Castle library) and `micro-ecc` as depicted in Figure 6. The difference is even more significant on less powerful devices such as wearables.

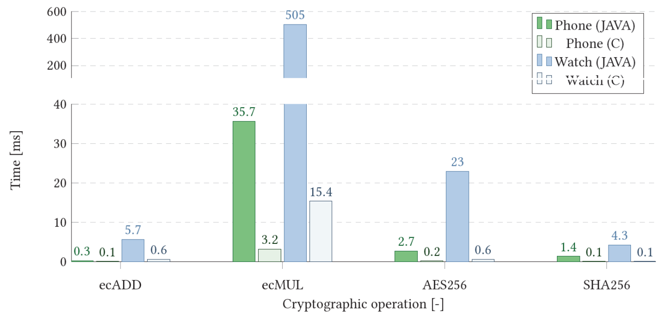

Figure 6: Speed comparison of JAVA and C libraries on Android devices (Xiaomi Redmi note 8 Pro - Green, Huawei Watch 2 - Blue).

The implementation results of Algorithms 1, 3, 4 on different IoT devices are shown in Table 3. The performance is measured in milliseconds, since the measurement of clock cycles is unavailable on the smart card platform and on wearables. Furthermore, the clock cycles are not meaningful for practical demonstration. We use the standard personal computer to simulate the server side. We let the smartphone act as MD while SDs where represented by smartwatches, microcontrollers, single board computers, and smartcards. The `micro-ecc` library was used in all implementations except the smartcard implementations. We used RIOT to run the same code on all microcontrollers. However, in the case of Arduino Nano we were not able to run the protocol using RIOT

Table 1: HW and SW specification of deployed devices.

Device	CPU/MCU	OS	RAM
	Computers and single board computers		
Raspberry Pi 4 Model B	ARM Cortex-A72	Raspberry Pi OS, Kernel v5.10	4.0 GB
Raspberry Pi Zero W	ARM11	Raspberry Pi OS, Kernel v5.10	512 MB
HP Pavilion 15	Intel i5-9300H	Windows 10	16 GB
	Smartphones		
Xiaomi Redmi note 8 Pro	Helio G90T	Android 10	6 GB
	Smartwatches		
HUAWEI Watch 2	Snapdragon 2100	Android Wear 2	768 MB
PineTime	ARM Cortex-M4F	RIOT 2021.04	64 KB
	Microcontrollers		
Arduino Nano	atmega328p	Arduino	2 KB
Arduino Due	AT91SAM3X8E	RIOT 2021.04	96 KB
ESP8266	LX106 RISC	RIOT 2021.04	64 KB
ESP32	LX6	RIOT 2021.04	520 KB
	Smartcards		
MultOS ML4	SC23Z018	MultOS 4.2	2.0 KB
JavaCard J3H145	P60D144	JCOP 3 v3.0.4	2.558 KB

Table 2: Cryptographic algorithms for AKA protocol.

Security	Strength	Elliptic curve	Cipher	Hash
Legacy*	80	SECP192	AES-GCM-128	SHA1
Sufficient	112	SECP224	AES-GCM-128	SHA224
Medium	128	SECP256	AES-GCM-128	SHA256
Higher	192	SECP384	AES-GCM-192	SHA384
The Highest	256	SECP512	AES-GCM-256	SHA512

Note: * We recommend not to use the Legacy parameters, since they are currently considered as insecure by NIST and ENISA.

due to the limited resources, so we reimplemented it using pure Arduino. The code for MultOS ML4 smartcard was written in MultOS assembly code and C language while the JavaCard J3H145 application was written in JavaCard programming language using Java Card Development Kit (JCDK) 3.0.4. Since the JavaCard does not support modular arithmetic operations nor EC operations, we had to implement them. In case of modular arithmetic operations, we implement them from scratch as software solution. In case of EC operations, we used hardware solution, i.e. we used smartcard coprocessor through `class javacard.security.KeyAgreement` and `ALG_EC_SVDP_DH_PLAIN_XY`, `ALG_EC_PACE_GM` algorithms.

Finally, we benchmark whole SSAKA implementation including a communication overhead. The implementation consists of one smartphone (Xiaomi Redmi note 8 Pro) which is used by client as a master device. The smartphone uses Near Field Communication (NFC) wireless technology to communicate with the server. We use smartwatch (HUAWEI Watch 2) as one secondary device needed to gain an access to the server and establish a secure communication channel. The smartwatch communicates with the smartphone through Bluetooth Low Energy (BLE) wireless communication channel. The Figure 7 shows our experimental results for different security strengths. The whole SSAKA protocol with deployed devices takes less than 600 ms for the 128-bit security strength (SECP256). The `Server` time (marked in red) shows time needed by server to generate its signature and very the user's proof. The `Smartphone` time (marked in green) shows time needed by smartphone to verify the server and generate user's proof including the NFC communication overhead. The `Smartwatch` time shows the time complexity of secondary device including BLE communication overhead.

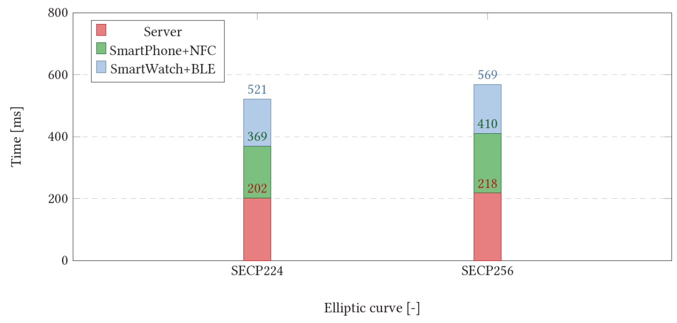

Figure 7: Time complexity of SSAKA for one secondary device and different elliptic curves.

7 CONCLUSION

In this paper, we proposed two novel AKA schemes, namely AKA and SSAKA. The basic AKA scheme is based on zero-knowledge-proof protocols and it is efficient even on constrained devices that are very often used in current IoT ecosystems. The SSAKA scheme extends our AKA to support multi-device and multi-factor authentication. Sharing the client's secret among more user devices increases the security strength of the algorithm. Examples of sharing parties are wearables, embedded microcontrollers, smartcards, or even the client him/herself using passwords or PIN codes. Both protocols' full security analysis is provided and implementation aspects are described in this paper. Our SSAKA can establish a secure communication channel in less than 600 ms for one secondary

Table 3: Performance of SSAKA algorithms on various devices.

Device	Product	Library	Entity	SECP224	SECP256
Personal Computer	HP Pavilion 15	micro-ecc	Server	202 ms	218 ms
Smartphone	Xiaomi Redmi note 8 Pro	micro-ecc	Master Device	25 ms	25 ms
Single Board Computer	Raspberry Pi 4 model B	micro-ecc	Secondary Device	3 ms	5 ms
Single Board Computer	Raspberry Pi Zero W	micro-ecc	Secondary Device	11 ms	19 ms
Smartwatch	HUAWEI Watch 2	micro-ecc	Secondary Device	39 ms	27 ms
Smartwatch	PineTime	micro-ecc	Secondary Device	182 ms	317 ms
Microcontroller	ESP8266	micro-ecc	Secondary Device	557 ms	1264 ms
Microcontroller	ESP32	micro-ecc	Secondary Device	276 ms	514 ms
Microcontroller	Arduino Due	micro-ecc	Secondary Device	222 ms	379 ms
Smartcard	MultOS ML4	-	Secondary Device	151 ms	163 ms
Smartcard	JavaCard J3H145	-	Secondary Device	968 ms	1054 ms

device and the 128-bit security strength. The protocol complexity is given by the computational capability of the most contained secondary device. This is due to the fact that the communication between the master device and secondary devices is run in parallel. As a next step, we will focus on adding privacy-enhancing features to this scheme.

ACKNOWLEDGMENTS

Research described in this paper was financed by the Technology Agency of the Czech Republic 'DELTA 2 Programme' under grant TM02000036. The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 813278 (A-WEAR: A network for dynamic wearable applications with privacy constraints, http://www.a-wear.eu/).

REFERENCES

[1] c2013 - 2021. RIOT - The friendly Operating System for the Internet of Things. https://www.riot-os.org/

[2] c2021. Micro-ecc. https://github.com/kmackay/micro-ecc

[3] Diego de Freitas Aranha and Conrado Porto Lopes Gouvêa. 2018. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[4] Amos Beimel. 2011. Secret-sharing schemes: a survey. In *International conference on coding and cryptology*. Springer, 11–46.

[5] Simon Blake-Wilson and Alfred Menezes. 1998. Authenticated Diffe-Hellman key agreement protocols. In *International Workshop on Selected Areas in Cryptography*. Springer, 339–361.

[6] Jan Camenisch and Markus Stadler. 1997. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*. Springer, 410–424.

[7] Liqun Chen and Caroline Kudla. 2003. Identity based authenticated key agreement protocols from pairings. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* IEEE, 219–233.

[8] Whitfield Diffie and Martin Hellman. 1976. New directions in cryptography. *IEEE transactions on Information Theory* 22, 6 (1976), 644–654.

[9] Jan Hajny, Petr Dzurenda, and Lukas Malina. 2016. Multi-Device Authentication using Wearables and IoT.. In *SECRYPT*. 483–488.

[10] Jan Hajny, Petr Dzurenda, and Lukas Malina. 2018. Multidevice Authentication with Strong Privacy Protection. *Wireless Communications and Mobile Computing* 2018 (2018).

[11] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.

[12] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. 2003. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography* 28, 2 (2003), 119–134.

[13] Ana Paula G Lopes, Lucas O Hilgert, Paulo RL Gondim, and Jaime Lloret. 2019. Secret sharing-based authentication and key agreement protocol for machine-type communications. *International Journal of Distributed Sensor Networks* 15, 4 (2019), 1550147719841003.

[14] Noel McCullagh and Paulo SLM Barreto. 2005. A new two-party identity-based authenticated key agreement. In *Cryptographers' Track at the RSA Conference*. Springer, 262–274.

[15] Reem Melki, Hassan N Noura, and Ali Chehab. 2019. Lightweight multi-factor mutual authentication protocol for IoT devices. *International Journal of Information Security* (2019), 1–16.

[16] Zeyad Mohammad, Ahmad Abusukhon, and Thaer Abu Qattam. 2019. A survey of authenticated Key Agreement Protocols for securing IoT. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. IEEE, 425–430.

[17] The Legion of the Bouncy Castle. 2013. Bouncy Castle Crypto APIs. http://www.bouncycastle.org.

[18] Tatsuaki Okamoto. 1992. Provably secure and practical identification schemes and corresponding signature schemes. In *Annual international cryptology conference*. Springer, 31–53.

[19] Sylvain Pasini and Serge Vaudenay. 2006. SAS-based authenticated key agreement. In *International Workshop on Public Key Cryptography*. Springer, 395–409.

[20] David Pointcheval and Jacques Stern. 2000. Security arguments for digital signatures and blind signatures. *Journal of cryptology* 13, 3 (2000), 361–396.

[21] Alavalapati Goutham Reddy, Ashok Kumar Das, Vanga Odelu, Awais Ahmad, and Ji Sun Shin. 2019. A privacy preserving three-factor authenticated key agreement protocol for client–server environment. *Journal of Ambient Intelligence and Humanized Computing* 10, 2 (2019), 661–680.

[22] Alavalapati Goutham Reddy, Eun-Jun Yoon, Ashok Kumar Das, Vanga Odelu, and Kee-Young Yoo. 2017. Design of mutually authenticated key agreement protocol resistant to impersonation attacks for multi-server environment. *IEEE access* 5 (2017), 3622–3639.

[23] Claus-Peter Schnorr. 1989. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*. Springer, 239–252.

[24] Mike Scott. 2002. Authenticated ID-based Key Exchange and remote log-in with simple token and PIN number. *IACR Cryptol. ePrint Arch.* 2002 (2002), 164.

[25] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[26] Kyungah Shim. 2003. Efficient ID-based authenticated key agreement protocol based on Weil pairing. *Electronics Letters* 39, 8 (2003), 653–654.

[27] Nigel P Smart. 2002. Identity-based authenticated key agreement protocol based on Weil pairing. *Electronics letters* 38, 13 (2002), 630–632.

[28] Yongge Wang. 2013. Efficient identity-based and authenticated key agreement protocol. In *Transactions On Computational Science Xvii*. Springer, 172–197.

[29] Zheng Yang, Junyu Lai, Yingbing Sun, and Jianying Zhou. 2019. A novel authenticated key agreement protocol with dynamic credential for WSNs. *ACM Transactions on Sensor Networks (TOSN)* 15, 2 (2019), 1–27.