

Security Protocols: Helping Alice and Bob to Share Secrets (COMPSEC.220)

Coursework II

Antonis Michalas
antonios.michalas@tuni.fi

September 6, 2022

SUBMISSION DEADLINE: 15.12.2022 AT 13:00

Important!

1. In case you do not get at least 40% in this coursework, you immediately fail the course. There will be no other chance to re-submit another coursework.
2. The coursework is **strictly personal**. If plagiarism is detected a penalty resulting to a mark of zero will be applied.

About this Coursework

This is a programming coursework that contains two different assignments and counts towards 40% of your overall course mark^a. You should choose one of the two available options.

In the first assignment, you are required to implement a Group Key Distribution algorithm based on Certificateless Public Key cryptography using the specifications from the publication provided.

While for the second assignment, you are required to develop a Symmetric Searchable Encryption (SSE) scheme using the information/specifications also provided.

Tutorials 6 and 7 provide basic templates that should help you complete either assignment.

Some Rules

- You should only **choose one** out of the two assignments.
- Programming Language: For both assignments, you can use any programming language of your choice.
- Dataset: To test your SSE implementation, you should use the dataset provided on <https://zenodo.org/record/3360392>

What you are expected to submit

1. A ZIP file with well structured and commented code (including your database);
2. A Readme file describing how to run your code;
3. A URL (a single TXT file or it can be part of the Readme) with a running demo (screenshot) of your implementation.
4. A PDF file with an analysis of the experiments you conducted. This should follow exactly the experimental results of the given paper.

^asee the official course description for course marking details

ASSIGNMENTS

ASSIGNMENT 1 – (100 MARKS)

Your task is to implement the Group Key Distribution protocol illustrated in the paper given below. The goal is to recreate the "Experimental results" section of the paper.

Note: All experiments should be conducted on a standard PC. Do not worry about using a resource-constrained device. You can run multiple instances of your implementation on the same machine to simulate multiple devices trying to establish a secure session.

Arrows in a Quiver: A Secure Certificateless Group Key Distribution Protocol for Drones

Eugene Frimpong^[0000–0002–4924–5258], Reyhaneh Rabbaninejad^[0000–0002–4907–2844], and Antonios Michalas^[0000–0002–0189–3520]

Tampere University, Tampere 33720, Finland
{eugene.frimpong, reyhaneh.rabbaninejad, antonis.michalas}@tuni.fi
<https://research.tuni.fi/nisec/>

Abstract. Drone-based applications continue to garner a lot of attention due to their significant potential in both commercial and non-commercial use. Owing to this increasing popularity, researchers have begun to pay attention to the communication security requirements involved in deploying drone-based applications and services on a large scale, with particular emphasis on group communication. The majority of existing works in this field focus on the use of symmetric key cryptographic schemes or group key agreement schemes. However, in this paper, we propose a pairing-free certificateless group authenticated key distribution protocol for drone-based applications which takes into consideration drones with varying computational resources. The proposed scheme ensures key freshness, group key secrecy, forward secrecy, and backward secrecy while ensuring that the scheme is lightweight enough to be implemented on very resource-constrained drones or smart devices. We extensively prove the security of our scheme and demonstrate its real-world applicability by evaluating its performance on three different kinds of drone boards (UP Xtreme i7 board, SamL11-Xpro board, and a Zolertia Re-mote Revb board).

Keywords: Certificateless Public Key Cryptography · Group Key Distribution · Drones

1 Introduction

Unmanned Aerial Vehicles (UAV) are gaining popularity in the Industries, Academia, and peoples’ personal lives at a rapid and accelerating pace. Big organizations, like Uber and Amazon, are constantly hinting at offering drone-based services such as package and food delivery [1]. Additionally, drones have been used for other consumer-related activities such as aerial photography, landscape surveying, and in some cases, delivering medical supplies to remote places. These devices come equipped with various capabilities and features – from high definition cameras to temperature sensors. Although drones are expected to offer numerous benefits to consumers and companies, the proliferating adoption of drone-based services presents a myriad of security concerns and requirements,

chief among them being secure communication [2,3]. Secure communication in drones centres around securing the communication channel between drones and their command centre, between individual drones, or groups of drones.

In this paper, we propose a pairing-free certificateless authenticated group key distribution protocol for drone-based applications. Early group key management schemes predominantly focused on symmetric-key based approaches where symmetric session keys were pre-installed on devices. However, this approach proved not be scalable for Wireless Sensor Networks (WSNs), a classification that applies to drones [4]. Subsequently, improvements to Elliptic Curve Cryptographic (ECC) primitives have led to an increased adoption of Public Key Cryptographic (PKC) schemes for resource-constrained environments [5]. Unfortunately, ECC schemes with certificates and pairing-based operations, incur additional certificate and computational overhead. To mitigate the limitations related to certificate overhead, many Certificateless Group Key Agreement (CL-GKA) schemes [6,7,8] have been proposed. However, these schemes are based on Group Key Agreement (GKA) protocols (all group members collaboratively calculate the group session key without depending on a trusted party), as compared to the Group Key Distribution (GKD) model we follow. There have been many arguments for GKA over GKD, such as the security of GKD protocols being broken when the group manager is compromised as well as its inappropriateness for distributed environments where a trusted authority or central authority is unavailable [9].

Contrary to these points, we argue that, for a drone-based application such as a Smart City consisting of different drones with varying computational resources and smart devices with equally varying resources, a GKA approach is inefficient. To support our argument, we consider a case study involving a drone team leader who receives mission plans and tasks from a central point and a group of edge drones deployed to accomplish the tasks allocated by the drone team leader. For our case study, the edge drones are assumed to have limited computational resources, with the team leader, on the other hand having considerably high ones. In such a case, a GKA approach is inefficient and is not scalable as the number of edge drones increases. This is due to the fact that in order to compute a session key, all devices are required to be online – which also introduces an additional communication overhead. Our protocol provides an efficient group authenticated key distribution protocol suitable for the case study described. Additionally, it can also be extended for environments consisting of resource-constrained smart devices deployed to sense and generate data.

Contributions: The contributions for this paper are summarized below:

- C1.** We propose a pairing-free certificateless group authenticated key distribution protocol for multi-drone applications and environments. The security of existing identity-based public key solutions is impacted by the use of a fully trusted KGC (i.e., Key Escrow problem). In our proposed scheme, the KGC is not fully trusted.
- C2.** We provide a comprehensive security analysis to prove the security of the proposed protocol.

- C3.** Finally, we implement and evaluate the performance of the proposed protocol on three different device platforms to demonstrate its benefits and applicability.

2 Related Work

One of the early key distribution schemes was introduced by Tian et al [10]. They presented a scheme based on Identity-based cryptography (ID-PKC) and bilinear pairings. Traditional ID-PKC suffer from the key escrow problem while the computational costs required for pairing operations are considerably higher than standard ECC operations such as EC point multiplications on resource-constrained devices. Kumar et al. [11] also proposed an efficient centralized group key distribution protocol based on the RSA public key cryptosystem, with particular emphasis on reducing the computation costs and storage complexity at the Key Server (KS). The scheme offers both *forward* and *backward secrecy* - an essential requirement [9,12] for any secure group key distribution protocol. A few notable drawbacks with this scheme are the same key escrow problem, certificate management overhead, and the computational complexity of the RSA scheme on resource-constrained devices and environments [13]. As a result of the key escrow and certificate management overhead, several certificateless public key cryptography schemes have been proposed [6,7,8,14,15,16], both for one-to-one communication and group-based communication instances.

In [6], authors propose a certificateless GKA scheme for unmanned aerial vehicles. Similar to majority of key agreement protocols [17,18,19,20,21], this protocol requires that each user contributes to the generation of the group key by way of a series of key establishment requests. At the end of the final round of the protocol, each user generates a similar session key. This work along with other certificateless schemes such as [14] and [7] ensure mutual key agreement, key escrow elimination, joint key control and key freshness. However, this scheme incurs relatively high computational burden at each user based on the pairing-based computations and does not consider a dynamic group where members of the group can join or leave a group. Similar certificateless key agreement schemes such as [16] and [15] also do not consider group environments.

More recently, a blockchain-based mutual healing group key distribution scheme was proposed in [22]. In this work, the Ground control Station (GCS) for the drones builds a private blockchain where the distributed group keys generated by the GCS as well as a list of membership certificates are recorded. The GCS acts as the KS for this scheme and uses the blockchain to record transactions. Transactions, in the context of this scheme, are instances when members leave or join the group. Although authors prove that the proposed scheme is resistant to various attacks as compared to other mutual healing schemes [23,24,10], it poses significant computational overhead resulting from constant interaction with the blockchain.

To design an efficient and resource friendly protocol, AinQ, our pairing-free certificateless key distribution protocol, uses a Key Generation Center (KGC)

to distribute partial private and public keypairs to all users. Our scheme utilizes a hybrid encryption for multiple users and combines a data encapsulation and a key encapsulation mechanism to distribute the group session key. Additionally, the computational burden rests primarily on the team leader. As such, AinQ can be extended to an IoT environment with significantly resource-constrained devices.

3 System Model

Our setup consists of four entities: *(i)* Key Generation Center, *(ii)* Cloud Service Provider (CSP), *(iii)* Edge Drones, and *(iv)* Team Leaders.

1. **KGC:** This is a semi-trusted entity responsible for generating and setting the system parameters for the complete run of the protocol. The KGC generates partial private and public key pairs for each registered drone during the protocol initialization phase.
2. **CSP:** We assume the existence of a CSP, an abstract external platform that consists of cloud hosts operating virtual machines that communicate through a network. The CSP will be the final destination of messages aggregated by the set of drones within our environment. Specific capabilities and features of the CSP are beyond the focus of this paper and as such are not discussed in detail. Our proposed scheme is independent of the underlying cloud platform.
3. **Edge Drones:** Let $\mathcal{D} = \{d_1, \dots, d_n\}$ be the set of all edge drones in our environment. Each drone is equipped with a number of sensors to monitor and report on sensed events. Each d_i accepts mission tasks and securely stores and updates mission data so that no adversary can learn anything.
4. **Team Leaders:** Let $\mathcal{Q} = \{q_1, \dots, q_m\}$ be the set of drones elected as Team Leaders in our protocol. Each drone team leader maintains a group list which contains the group members and their respective public keys. This group list is updated when a drone joins or leaves the group. Each team leader accepts missions from the CSP and assigns individual tasks to members of its group. Note that a team leader is assumed to be a more powerful drone with far more computational resources compared to a regular edge drone.

4 Arrows in a Quiver (AinQ)

In this section, we present AinQ, which constitutes the core of our contribution. AinQ's description is divided into two parts:

1. The construction of a scheme containing algorithms for individual and group key generation, key retrieval and re-keying.
2. A protocol showing how our scheme can be effectively used to allow drones to form groups and securely agree on secret keys that will allow them to securely exchange information over an encrypted channel.

4.1 AinQ Scheme

For the purposes of AinQ, we extend the functionalities of eCLSC-TKEM [16] with the **GenGroupKey**, **KeyRetrieval**, and **Re – Key** algorithms to support group key distribution (using a form of Multiple-Recipient/Multiple-Message Public Key Encryption (MR-MM-PKE) [25]). In total, our scheme consists of the following seven probabilistic algorithms.

Setup: This algorithm is run by the KGC to generate the system parameters for the scheme and a master secret key. The algorithm takes as input a security parameter $\lambda \in \mathbb{Z}^+$, and outputs the system parameters Ω , and the KGC's master secret key msk . Given λ , KGC executes the following steps:

- Step 1.** Chooses a λ -bit prime q and a point P on the curve G_q .
- Step 2.** Chooses msk as $x \in \mathbb{Z}^*$.
- Step 3.** Computes the corresponding public key as $P_{\text{pub}} = xP$.
- Step 4.** Chooses the following cryptographic hash functions where n is the key length of the symmetric key encryption scheme:
 - $H_0 : \{0, 1\}^* \times G_q^2 \times \{0, 1\}^* \rightarrow \mathbb{Z}_q^*$,
 - $H_1 : G_q^3 \times \{0, 1\}^* \times G_q \rightarrow \{0, 1\}^n$,
- Step 5.** Publishes the system parameters $\Omega = \{G_q, P_{\text{pub}}, P, H_0, H_1\}$.

GenSecretValue: Each edge drone $d_i \in \mathcal{D}$ and team leader run this algorithm to generate a secret value and a public key. The algorithm takes as input the system parameters Ω generated in the Setup algorithm, the drone identity d_i , and outputs a secret value x_i along with a corresponding public key P_i . Given Ω , d_i executes the following steps:

- Step 1.** Chooses a secret value $x_i \in \mathbb{Z}^*$,
- Step 2.** Computes the corresponding public key as $P_i = x_i P$.

GenPartialKey: The KGC runs this algorithm to generate a partial key for all registered drones. It takes as input the drone's identity d_i , its public key P_i , and the master secret key x . On a successful run, **GenPartialKey** outputs the partial private and public keys for d_i . Given P_i , the KGC executes the following steps:

- Step 1.** Chooses $r_i \in \mathbb{Z}^*$
- Step 2.** $R_i = r_i \cdot P$
- Step 3.** $s_i = r_i + xH_0(d_i, R_i, P_i) \mod q$

FullKeyGen: Each registered drone runs this algorithm to generate its full private key sk_i and public key pk_i . The algorithm takes as input the drone's secret value x_i , partial secret key s_i , public key P_i and partial public key R_i . On successful run, it returns the drone's full private and public key pair.

GenGroupKey: This algorithm is run by a designated team leader $q_k \in \mathcal{Q}$ to generate a symmetric group session key K_g for the group. Given a group list

$GL = \{d_1, \dots, d_h\}$ containing a list of valid group members and their respective public keys $pk_i, i \in \{1, \dots, h\}$, q_k generates a list of ciphertexts $C_i, i \in \{1, \dots, h\}$. The algorithm takes as input the group list GL and the valid time period t_g . Given GL, q_k executes the following steps:

Step 1. Chooses $K_g \in \mathbb{Z}^*$ and $l_k \in \mathbb{Z}^*$ at random.

Step 2. Computes $V = l_k \cdot P$.

Step 3. Parses pk_i as (R_i, P_i) for all $d_i \in GL$.

Step 4. For each pk_i :

- $Y_i = R_i + H_0(d_i, R_i, P_i) \cdot P_{pub} + P_i$.
- $T_i = l_k \cdot Y_i$.
- $C_i = K_g \oplus H_1(V, T_i, q_k, pk_k, d_i, pk_i, t_g)$.

Step 5. Outputs $(V, C_1, C_2, \dots, C_h, t_g)$.

KeyRetrieval: This is the key retrieval algorithm run by each drone $d_i \in GL$ to obtain the group key K_g generated by q_k in **GenGroupKey**. Given the broadcast message containing the list of ciphertexts $(V, C_1, C_2, \dots, C_h)$, and the respective private key and public key of the recipient drone, d_i retrieves the group key K_g . We denote this by: $K_g \leftarrow \text{KeyRetrieval}(V, C_1, C_2, \dots, C_h, sk_i, pk_i)$. Given $(V, C_1, C_2, \dots, C_h)$, each $d_i \in GL$ executes the following steps:

Step 1. Computes $T_i = (s_i + x_i) \cdot V$

$$(s_i + x_i) \cdot l_k \cdot P = l_k \cdot Y_i$$

Step 2. $K_g = C_i \oplus H_1(V, T_i, q_k, pk_k, d_i, pk_i, t_g)$.

Re – Key: This algorithm is run by the team leader q_k whenever a new drone joins the group, an existing member leaves, or the an existing group key expires and a new one has to be issued. Given an updated group list $GL = \{d_1, \dots, d_h\}$ containing an up-to-date information on group members, q_k generates a new group key K_g' . We denote this by: $(V, C'_1, C'_2, \dots, C'_h) \leftarrow \text{ReKey}(GL)$. Given the updated GL, q_k executes the following steps:

Step 1. Chooses a new group key $K_g' \in \mathbb{Z}^*$

Step 2. If d_i is a new member:

- Parse pk_i as (R_i, P_i) for $d_i \in GL$.
- $Y_i = R_i + H_0(d_i, R_i, P_i) \cdot P_{pub} + P_i$.
- $T_i = l_k \cdot Y_i$.

Step 3. $C'_i = K_g' \oplus H_1(V, T_i, q_k, pk_k, d_i, pk_i, t_g)$

Step 4. Outputs $(V, C'_1, C'_2, \dots, C'_h)$.

4.2 AinQ Protocol

The proposed protocol is divided into 3 phases; (i) Setup and Initialization, (ii) Key Generation and Retrieval, and (iii) Group Re-keying. To provide a detailed and comprehensive description of each phase, we consider a drone-based scenario consisting of an elected drone group leader and a number of edge drones in its group. In our assumed scenario, the elected drone team leader q_k wishes to distribute a group key K_g to all edge drones belonging to GL in the presence of a KGC. Furthermore, we assume that all drones have a maximum flight time of t_g and are stored in a secure location when not on a mission.

AinQ - Setup and Initialization The KGC runs the **Setup** algorithm at the beginning of the protocol to generate a master secret key and system parameters. The algorithm returns the system parameters, Ω , and the master secret key x . These system parameters are public and accessible to each registered entity partaking in the protocol. Each registered drone runs the **GenSecretValue** algorithm to generate a secret value and a corresponding public key. On successful run of this algorithm, the drones send their identity and public key to the KGC in order to receive partial private and public keys valid for the length of their flight. The KGC runs the **GenPartialKey** algorithm and returns to each drone the partial private and public key pair. All communication in this phase of the protocol occurs before the drones leave for a mission and is assumed to be over a secure channel.

Upon receiving the partial private/public keypair, each drone runs the **GenPrivKey** and **GenPubKey** to generate a full public/private key pair. We assume that each drone makes its public key available to all other drones.

AinQ - Key Generation and Retrieval In this phase of AinQ, the team leader q_k first generates a random number r_1 , and runs the **GenGroupKey** algorithm to generate the symmetric group key K_g and the list of ciphertexts (C_1, C_2, \dots, C_h) . K_g that will be used to secure all ensuing communication between the group members as well as with the team leader.

On successful run of the **GenGroupKey** algorithm, the team leader sends the following broadcast message to drones in the network: $m_1 = \langle r_1, V, C_1, C_2, \dots, C_h, q_k, t_g, \sigma_{q_k} \rangle$ where $\sigma_{q_k} = \text{sig}_{sk_k}(r_1 || V || K_g)$. Upon receiving m_1 , each registered drone executes the **KeyRetrieval** algorithm to retrieve the group key K_g . The freshness and integrity of m_1 is verified using the team leader's public key and the generated group key. The protocol is aborted if the signature verification process fails. Figure 4.1 provides an illustration of this phase.

AinQ - Group Re-Key The team leader q_k runs the **Re - Key** algorithm in this phase to generate a new group key K_g' whenever a new drone joins its group or an existing drone leaves the group. The re-keying process ensures that AinQ is both forward and backward private. When a drone leaves or joins a group, the leader updates the group list GL' , generates a new random number r_2 , and broadcasts a new message m_2 to the network. $m_2 = \langle r_2, V, C'_1, C'_2, \dots, C'_h, q_k, t_g', \sigma'_{q_k} \rangle$ where $\sigma'_{q_k} = \text{sig}_{sk_k}(r_2 || V || K'_g)$.

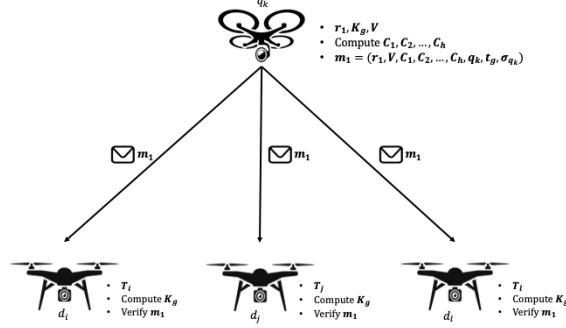


Fig. 4.1. Key Generation and Retrieval Phase

5 Security Analysis

In this section, we proceed to prove the security of our construction in the presence of a malicious adversary \mathcal{A} , who can be an outside adversary— which covers a variety from a passive eavesdropper who just listens to the network to a malicious entity who has captured some drones—, or inside adversaries including a corrupt KGC and a revoked user. We begin by describing the main security properties that a group key distribution scheme should satisfy (5) and follow this up with the necessary security definitions that we consider for our threat model (5).

Security Requirements: Consider a group where edge drones dynamically join or leave. Furthermore, let $\mathcal{K} = \{K_g^0, \dots, K_g^s\}$ be the set of sequential group keys generated during s successive sessions. Below we provide a list of the main security properties that a GKD scheme should satisfy.

1. *Key Freshness:* A GKD scheme has this property if it guarantees a key to be new, thus preventing the reuse an old key by an adversary.
2. *Group Key Secrecy:* A GKD scheme must guarantee that a session key is only known to legitimate drones. This means that extracting a session key $K_g^i \in \mathcal{K}, i \in [0, s]$ is computationally infeasible for an adversary.
3. *Forward Secrecy:* Assume an adversary possesses a consecutive subset of session keys (e.g., $\{K_g^0, K_g^1, \dots, K_g^i\}$). This property guarantees that he can learn nothing about a future session key K_g^j , for all $i < j$. Therefore, a revoked drone cannot discover future session keys.
4. *Backward Secrecy:* Assume an adversary possesses a consecutive subset of session keys (e.g., $\{K_g^i, K_g^{i+1}, \dots, K_g^j\}$). This property guarantees that he can learn nothing about a past session key K_g^l , for all $l < i < j$. Therefore, a newly joined drone cannot discover previous session keys.

Security Model: We now formally define indistinguishability against adaptive chosen ciphertext attack (IND-CCA2) through the following game between an

adversary \mathcal{A} (this can be an outside adversary, a corrupt KGC, or a revoked user) and a challenger \mathcal{B} .

Ainq-IND-CCA2 Game

- Challenger \mathcal{B} runs the **Setup** algorithm to generate msk , the corresponding public key P_{pub} , and system parameters Ω . \mathcal{B} then forwards Ω and P_{pub} to \mathcal{A} and keeps msk confidential. In case \mathcal{A} is a corrupt KGC, msk is also sent to \mathcal{A} .
- Adversary \mathcal{A} can make the following queries to the challenger. In case \mathcal{A} is a revoked user, the run time of the operations executed by \mathcal{B} is less than the challenge time period.
 1. *GenSecretValue* Query. Adversary \mathcal{A} queries the secret value and the corresponding public key of a specified drone. \mathcal{B} runs the **GenSecretValue** algorithm and forwards the output to \mathcal{A} . Note that we exclude a corrupt KGC from these queries.
 2. *GenPartialKey* Query. To respond to a query on the partial private and public keypair of a specified drone from \mathcal{A} , \mathcal{B} runs the **GenPartialKey** algorithm with msk and the drone's public key as inputs, and forwards the output to \mathcal{A} .
 3. *GenGroupKey* Query. Adversary \mathcal{A} sends a query to $\mathcal{O}_{\text{GenGroupKey}}$ oracle by giving as input the group list GL , identity q_k of the team leader, and the valid time period t_g . Using the key for group of drones GL generated by team leader q_k for time period t_g , \mathcal{B} runs **GenGroupKey** algorithm and forwards the output to \mathcal{A} .
 4. *KeyRetrieval* Query. Adversary \mathcal{A} queries $\mathcal{O}_{\text{KeyRetrieval}}$ oracle to extract a group key from the broadcast message. \mathcal{B} runs **KeyRetrieval** algorithm and forwards the output to \mathcal{A} . Note that we exclude a corrupt KGC from these queries.
 5. *Re – Key* Query. Adversary \mathcal{A} sends a query to $\mathcal{O}_{\text{Re-Key}}$ oracle by giving as input an updated group list GL , the team leader's identity q_k , and the valid time period t_g . \mathcal{B} runs **Re – Key** algorithm and forwards the output to \mathcal{A} .
- At the end of query phase, \mathcal{A} submits challenge inputs including group list GL^* , team leader identity q_k^* , and a valid time period t_g^* , and two session keys K_g^0, K_g^1 . \mathcal{A} may not have made *FullKeyGen* queries on any of the identities in GL^* and q_k^* by querying both $\mathcal{O}_{\text{GenSecretValue}}$ and $\mathcal{O}_{\text{GenPartialKey}}$ oracles. Also, \mathcal{A} may not have made *KeyRetrieval* query on tuple (GL^*, q_k^*, t_g^*) in the query phase. In case \mathcal{A} is a revoked user, the condition $t_g^* > t_R$ must also hold, where t_R is the revocation time. That is, in the challenge time period, \mathcal{A} has no access to new information. \mathcal{B} picks a random $b \in \{0, 1\}$ and runs $(V^*, C_1^*, C_2^*, \dots, C_h^*) \leftarrow \text{GenGroupKey}(GL^*, q_k^*, t_g^*, K_g^b)$, where $C_i^* = K_g^b \oplus H_1(V^*, T_i, q_k^*, pk_k^*, d_i^*, pk_i^*, t_g^*)$. Finally, \mathcal{B} sends $(V^*, C_1^*, C_2^*, \dots, C_h^*)$ as challenge to \mathcal{A} .

- Excluding the case where \mathcal{A} is a revoked user, \mathcal{A} can continue the query phase by adaptively making a polynomially bounded number of queries. \mathcal{A} may not make *FullKeyGen* queries on any identities in GL^* and q_k^* by querying both $\mathcal{O}_{GenSecretValue}$ and $\mathcal{O}_{GenPartialKey}$ oracles. Also, \mathcal{A} may not make *KeyRetrieval* query on same group list GL^* , team leader identity q_k^* , and time period t_g^* . Finally, \mathcal{A} outputs a bit b' and wins the game if $b' = b$.

Definition. AinQ is IND-CCA2 secure if any probabilistic polynomial-time adversary \mathcal{A} has at most negligible advantage in the above security game. \mathcal{A} 's advantage is defined as below:

$$Adv^{IND-CCA2}(\mathcal{A}) = |Pr[b' = b] - \frac{1}{2}|. \quad (5.1)$$

5.1 Security Proof

Below, we provide the formal security proof for AinQ which relies on the hardness of decisional Diffie–Hellman problem.

Definition: Decisional Diffie–Hellman (DDH) Assumption. Given a prime q and a generator P on the curve G_q , for randomly and independently chosen $a, b \in \mathbb{Z}_q$, the value abP is indistinguishable from a random element in G_q . Formally, for each probabilistic polynomial-time adversary \mathcal{A} which is given $(D_1 = aP, D_2 = bP)$ and a candidate solution D_3 , \mathcal{A} 's advantage to distinguish whether $D_3 = abP$ or whether D_3 was chosen at random from G_q is negligible. In other words, for any probabilistic polynomial-time algorithm \mathcal{A} , we have:

$$\begin{aligned} &|Pr[\mathcal{A}_{DDH}(G_q, P, aP, bP, cP) = 1] \\ &- Pr[\mathcal{A}_{DDH}(G_q, P, aP, bP, abP) = 1]| \leq \text{negl}(\lambda), \end{aligned}$$

where $a, b, c \in \mathbb{Z}_q$ are chosen at random.

Theorem 1. AinQ is IND-CCA2 secure under DDH assumption in the random oracle model.

Proof. As noted in the security model, we consider three types of adversaries: an outside adversary – which covers a variety from a passive eavesdropper who just listens to the network to a malicious entity who has captured some drones–, and inside adversaries including malicious KGC and a revoked user. Here we formally prove AinQ security against an outside adversary. Security proofs against malicious KGC and revoked user follow same arguments as proof below, which are omitted due to the space limitation.

–**Security Against Outside Adversary:** extracting a session key is computationally infeasible for an outside adversary. To show this, we prove that if an outside adversary \mathcal{A} has a non-negligible advantage in IND-CCA2 game, then there exists an algorithm \mathcal{B} that solves the DDH problem with overwhelming probability.

Setup. Given a DDH challenge $(D_1 = aP, D_2 = bP, D_3)$, \mathcal{B} sets the public key $P_{pub} = D_1$ and forwards it to \mathcal{A} . Here, the virtual master secret key msk is equal to $x = a$.

GenSecretValue Query. To answer a query on secret value of drone d_i submitted by \mathcal{A} , \mathcal{B} chooses random $x_i \in \mathbb{Z}^*$ and sets the corresponding public key as $P_i = x_i P$. Then, \mathcal{B} sends (x_i, P_i) to \mathcal{A} and also saves the pair (x_i, P_i) into a table T_{d_i} .

GenPartialKey Query. To answer a query on the partial key of a drone d_i submitted by \mathcal{A} , since \mathcal{B} does not possess msk , he generates $s_i = r_i + xH_0(d_i, R_i, P_i) \bmod q$ by controlling the output of H_0 as follows. \mathcal{B} chooses random $s_i \in \mathbb{Z}^*$ as the queried partial key. \mathcal{B} also selects random $c_i \in \mathbb{Z}^*$ as the output of $H_0(d_i, R_i, P_i)$ and computes $R_i = s_i P - c_i D_1$. Finally, \mathcal{B} checks if $(\{d_i, R_i, P_i\}, \cdot)$ is an entry in table T_{H_0} ; if it is so, the random c_i assigned to $H_0(d_i, R_i, P_i)$ is not correct and the game aborts. Otherwise, \mathcal{B} saves $(\{d_i, R_i, P_i\}, c_i)$ in table T_{H_0} and outputs the queried partial key as (s_i, R_i) which is also recorded in table T_{d_i} .

Hash Query. To answer H query on input a_i , \mathcal{B} first checks previously queried values in table T_H . If there is the same entry in T_H , he outputs the corresponding value. Otherwise, he outputs a random value $c_i \in \mathbb{Z}^*$ and saves (a_i, c_i) in T_H .

GenGroupKey Query. Adversary \mathcal{A} sends GL, q_k, t_g to query $\mathcal{O}_{GenGroupKey}$ oracle. To answer this query, \mathcal{B} chooses $K_g \in \mathbb{Z}^*$ and $l'_k \in \mathbb{Z}^*$ at random, and computes $V = l'_k \cdot D_2$. Next, \mathcal{B} extracts $\text{pk}_i = (R_i, P_i)$ from table T_{d_i} , for all $d_i \in GL$. Note that if table T_{d_i} for a drone $d_i \in GL$ was empty, \mathcal{B} generates the corresponding values by calling $\mathcal{O}_{GenSecretValue}$ and $\mathcal{O}_{GenPartialKey}$ oracles. For all pk_i , \mathcal{B} computes $T_i = l'_k s_i \cdot D_2 + l'_k c_i \cdot D_3 + l'_k x_i \cdot D_2$ and $C_i = K_g \oplus H_1(V, T_i, q_k, \text{pk}_k, d_i, \text{pk}_i, t_g)$. Finally, \mathcal{B} outputs $(V, C_1, C_2, \dots, C_h, t_g)$ as response to the query.

KeyRetrieval Query. Adversary \mathcal{A} queries \mathcal{B} to extract group key from a broadcast message $(V, C_1, C_2, \dots, C_h, t_g)$. \mathcal{B} extracts $\text{sk}_i = (s_i, x_i)$ from table T_{d_i} , for one d_i in group list GL corresponding to the broadcast message. \mathcal{B} then computes $T_i = (s_i + x_i) \cdot V$ and $K_g = C_i \oplus H_1(V, T_i, q_k, \text{pk}_k, d_i, \text{pk}_i, t_g)$ to retrieve the session key and forwards K_g to \mathcal{A} .

Re-Key Query. Adversary \mathcal{A} sends an updated group list GL , team leader q_k , and valid time period t_g to query \mathcal{O}_{Re-Key} oracle. To answer this query, \mathcal{B} performs same process as he did in GenGroupKey Query except that the T_i values for old drones can be reused from previous runs. Finally, the output is forwarded to \mathcal{A} .

Challenge. At the end of query phase, \mathcal{A} submits challenge inputs including group list GL^* , team leader identity q_k^* , and a valid time period t_g^* , and two session keys K_g^0, K_g^1 . If the conditions described in AinQ-IND-CCA2 game hold, \mathcal{B} (1) picks a random $b \in \{0, 1\}$ (2) runs GenGroupKey Query on input $(K_g^b, GL^*, q_k^*, t_g^*)$ to generate $(V^*, C_1^*, C_2^*, \dots, C_h^*)$ (3) sends it as challenge to \mathcal{A} .

Response. \mathcal{A} can run another query phase by adaptively making a polynomially bounded number of queries which must meet the conditions described in Ainq-IND-CCA2 game. Finally, (1) \mathcal{A} outputs a bit b' (2) \mathcal{B} responds to the DDH challenge by outputting 1 if $b' = b$, and 0 otherwise.

Analysis. The probability of aborting in the above game, is equal to the probability of collision in H_0 which is at most $q_H/2^\lambda$, where q_H is the total number of queries to H_0 . So, the probability that \mathcal{A} wins the game is $\epsilon(\lambda)(1 - q_H/2^\lambda)$. Regarding \mathcal{B} 's response, two cases can be considered:

Case 1. The DDH challenge given to \mathcal{B} is generated by randomly choosing $a, b, c \in \mathbb{Z}_q$, and setting $D_1 := aP$, $D_2 := bP$, and $D_3 := cP$. In this case, D_3 is a random element in G_q and thus $T_i^* = l_k^* s_i \cdot D_2 + l_k^* c_i \cdot D_3 + l_k^* x_i \cdot D_2$ is uniformly distributed in G_q . Therefore, the view of \mathcal{A} on the challenge ciphertext $C_i^* = K_g^b \oplus H_1(V^*, T_i^*, q_k^*, pk_k^*, d_i, pk_i, t_g^*)$, is distributed exactly as \mathcal{A} 's view in one-time pad (OTP). Since \mathcal{B} outputs 1 exactly when the output b' of \mathcal{A} is equal to b , we have that:

$$\begin{aligned} & Pr[\mathcal{B}_{DDH}(G_q, P, aP, bP, cP) = 1] \\ &= (1 - q_H/2^\lambda) \cdot Pr[\mathcal{A}_{OTP}(b = b')] \\ &= \frac{1}{2} \cdot (1 - q_H/2^\lambda). \end{aligned}$$

Case 2. The DDH challenge given to \mathcal{B} is generated by randomly choosing $a, b \in \mathbb{Z}_q$, and setting $D_1 := aP$, $D_2 := bP$, and $D_3 := abP$. In this case, the view of \mathcal{A} on the challenge ciphertext $C_i^* = K_g^b \oplus H_1(V^*, T_i^*, q_k^*, pk_k^*, d_i, pk_i, t_g^*)$ is distributed exactly as \mathcal{A} 's view in Ainq. Since \mathcal{B} outputs 1 exactly when the output b' of \mathcal{A} is equal to b , we have that:

$$\begin{aligned} & Pr[\mathcal{B}_{DDH}(G_q, P, aP, bP, abP) = 1] \\ &= (1 - q_H/2^\lambda) \cdot Pr[\mathcal{A}_{Ainq}(b = b')] \\ &= \left(\frac{1}{2} + \epsilon(\lambda)\right) \cdot (1 - q_H/2^\lambda). \end{aligned}$$

Therefore, \mathcal{B} 's advantage in solving the DDH challenge is:

$$\begin{aligned} & |Pr[\mathcal{B}_{DDH}(G_q, P, aP, bP, cP) = 1] \\ & - Pr[\mathcal{B}_{DDH}(G_q, P, aP, bP, abP) = 1]| \\ &= \left|\frac{1}{2} - \left(\frac{1}{2} + \epsilon(\lambda)\right)\right| \cdot (1 - q_H/2^\lambda) \end{aligned}$$

which implies that if $\epsilon(\lambda)$ is non-negligible, then the probability of solving DDH problem $\epsilon(\lambda) \cdot (1 - q_H/2^\lambda)$ is non-negligible too, completing the proof. \square

Discussion. Now we consider how security requirements are satisfied by Ainq. (1) *key Freshness*: this requirement is trivially satisfied since each new session key is chosen uniformly at random from the key space making the event

of repetitious session keys unlikely to happen. (2) *Group Key Secrecy*: this is a trivial inference of Theorem 1. (3) *Forward Secrecy*: whenever a revocation happens, the team leader executes **Re – Key** algorithm to refresh session key and distributes it through the network. Since the refreshed session key K_g^j is chosen independent from all previous session keys $\{K_g^0, K_g^1, \dots, K_g^i\}$ known to the leaving drone, revoked drone’s view is exactly same as the view of an outside adversary. Therefore, a former drone cannot discover subsequent session keys. This can be also deduced from Theorem 1. (4) *Backward Secrecy*: whenever a new drone joins the group, the team leader executes **Re – Key** algorithm to refresh session key randomly and distributes it through the network. Since all the new session keys $\{K_g^i, K_g^{i+1}, \dots, K_g^j\}$ known to the new drone are chosen independent from a previous session key K_g^l , for all $l < i < j$, new drone’s view with respect to the prior session keys is exactly same as the view of an outside adversary. Therefore, based on Theorem 1, a new drone can learn about previous session keys only with negligible advantage.

6 Experiments

In this section, we evaluate the performance of the AinQ’s core functions and their impact on our target devices. For the purposes of this experiment, we implemented the protocol on devices that are considered to be commercially available. Our testbed was made up of the following boards:

- **Team Leader**: An UP Xtreme board equipped with an Intel Core i7-8665UE SoC, 16GB RAM, 64GB storage capacity and an Intel UHD Graphics 620 graphics card¹. We installed Ubuntu 20.04 on this board and utilized the MIRACL cryptographic library [26] to implement the proposed protocol.
- **Edge Drones**: To provide a comprehensive evaluation on resource-constrained devices, we considered two boards for this role. The Zolertia Re-Mote Revb board which comes equipped with a 32MHz ARM Cortex-M3 SoC with 512KB flash, and 32KB RAM², and the SAML11 Xplained Pro board with a 32MHz ARM Cortex-M23 SoC, 64KB flash, and 16KB SRAM³. Implementations for both boards were built on top of the RIOT [27] OS using the C25519 cryptographic library⁴ for RIOT.

6.1 Performance of Core Cryptographic Functions

In this phase of our experiments, we evaluated the performance of the proposed cryptographic functions by measuring their execution times. For each specific

¹ <https://up-board.org/up-xtreme/>

² <https://github.com/Zolertia/Resources/wiki/RE-Mote>

³ <https://www.microchip.com/Developmenttools/ProductDetails/DM320205>

⁴ <https://www.dlbeer.co.nz/oss/c25519.html>

entity, we focused on the functions it executes directly. For example, when evaluating the performance on the resource-constrained edge drone, we focused exclusively on the `GenSecretValue`, and `KeyRetrieval` functions. For the team leader, we focused on the `GenSecretValue`, `GenGroupKey`, and `Re – Key` functions.

Edge Drone During the course of these experiments, we observed that the performance of the proposed functions on the resource-constrained edge drones depended heavily on the number of EC point multiplications performed by the device. Based on the specifications of the chosen target devices, we noticed a considerable difference in the execution times. The SAML11-xpro executed an EC multiplication in approximately 4.782 seconds while the Zolertia Re-mote board used approximately 2.598 seconds. Subsequently, the SAML11-xpro executed the `GenSecretValue` in approximately 5.343 seconds and the `KeyRetrieval` function in approximately 4.783 seconds. The Zolertia Re-mote board on the other hand, executed the `GenSecretValue` in approximately 2.943 seconds and the `KeyRetrieval` function in approximately 2.613 seconds.

	SAML11-Xpro Zolertia Re-mote		
	EM	Time (sec)	Time (sec)
EC Multiplication	0	4.782	2.598
GenSecretValue	1	5.343	2.943
KeyRetrieval	1	4.783	2.613

Table 6.1. Edge Drone Performance

Table 6.1 provides an overview of the results of the experiments conducted on the edge drone. Each experiment was conducted 50 times with the average time recorded. From the results, we observe that the Zolertia Re-mote is almost twice as efficient as the SAML11 Xplained pro. The difference in the performance results was to be expected based on the resources available to each of the boards.

Team Leader The overall performance of our proposed protocol at the team leader is determined by the execution of the `GenGroupKey` and `Re – Key` functions. To this end, we measure the execution time of the `GenGroupKey` function for a varying number of edge drones, ranging from 1 to 2,000. When the number of edge drones was 1, `GenGroupKey` took approximately 0.66 ms to execute whereas when the number of edge drones was 2,000, it executed in approximately 0.72 seconds. We observed that as the number of edge drones in the group increased, the execution time increased in an efficient manner due to the re-use of the same V parameter for all drones. To be more precise, multiplying 0.66 ms by 2,000 drones resulted in approximately 1.32 seconds. Consequently, we conclude that the `GenGroupKey` algorithm achieved an execution time which was about 50% more efficient than the expected performance.

As stated in subsection 4.1, the `Re – Key` function is executed when an edge drone joins the group, leaves the group or the current group key expires. To this end, we measured the execution of the `Re – Key` function by performing two sets

of experiments. The first set focused on renewing an expired key. Similar to the experiments for the **GenGroupKey** function, we executed the function for a range of 1 to 2,000 edge drones. For the instance of only 1 drone, the function execution time was approximately 0.03 ms, while when the number of drones in the group was 2,000, the execution time was approximately 15.28 ms. Figure 6.1 shows the overall execution times of both the **GenGroupKey** and **Re – Key** functions when the number of edge drones ranged from 1 to 2,000.

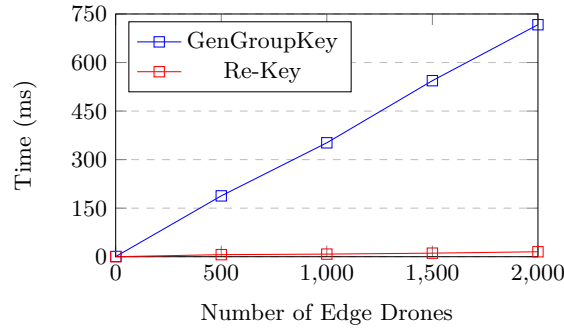


Fig. 6.1. Performance of the Team Leader

As a next step, we evaluated the performance of the **Re – Key** function when new edge drones join a group. To do this, we measured the execution of the **Re – Key** function when a varying number of new edge drones joined a group while maintaining a varying number of existing group members. When a new drone joins group containing 1 member, the **Re – Key** function takes approximately 0.47 ms to execute while when 1,000 users join a group which has 1,000 existing members, **Re – Key** function takes approximately 389.88 ms. Table 6.2 illustrates the results from these sets of experiments. It is worth mentioning that we exclude evaluations when a drone leaves a group as this is similar to a simple group key re-keying operation.

Comparison with Similar Works: One of our intentions during the experiments, was to compare our scheme with other similar works. We firmly believe that this would make our experimental evaluation more comprehensive. However, this proved to be difficult as similar works based on GKD techniques have not made their code publicly available and therefore we were unable to reproduce their results. However, since we believe that comparison with similar works can give valuable insights about the performance of our work, we attempted to compare our scheme to that presented in [6]. It is worth noting that the scheme in [6] is a certificateless GKA scheme (CL-GAKA) whereas AinQ is certificateless GKD scheme. As such, a comparison was *not* straightforward. For example, we compared the performance of the group key generation by the team leader in AinQ to the group key agreement by x number of users in CL-GAKA. For the

Existing Group Members	New Group Members	Time (ms)
1	1	0.47
1	100	32.48
1	500	185.79
1	1000	352.34
100	1	0.98
100	100	35.61
100	500	174.21
100	1000	355.44
500	1	0.59
500	100	39.89
500	500	183.71
500	1000	391.56
1000	1	8.40
1000	100	51.69
1000	500	209.42
1000	1000	389.88

Table 6.2. Group Re-key Function

purposes of this comparison, we implemented the CL-GAKA scheme on our UP Xtreme board using the PBC library⁵. The implementation was executed over a loopback interface (i.e. the same node emulates all clients considered during our experiments), with all measurements recorded over 50 iterations. We measured the performance of CL-GAKA's key agreement phase for three users, excluding the communication overhead, and observed an average time of 24.3ms. Additionally, we measured a user's performance when executing the computations needed to contribute to the key agreement phase and observed an average time of 5.5ms. On the other hand, the group key generation phase of AinQ takes approximately 1.22ms for three users while each user takes an average of 0.22ms to retrieve a received group key (summing up the group key retrieval time for three users equates to approximately 0.66ms). These results prove that the key distribution and pairing-free cryptographic approach employed by AinQ make it considerably more efficient than the pairing-based key agreement approach used by CL-GAKA. We acknowledge that the number of users considered for our implementation of CL-GAKA could have been more. However, practically implementing a GKA scheme with a large numbers of users was not a straightforward task; hence, supporting our argument that a GKD scheme is more scalable than a GKA scheme.

7 Conclusion

In this paper, a secure pairing-free certificateless group authenticated key distribution protocol is presented. The proposed scheme, AinQ, meets the requirements for a secure group key distribution protocol and considers multiple drones

⁵ <https://crypto.stanford.edu/pbc/>

with varying resource constraints. AinQ has been proven efficient for a group with up to 2,000 edge drones when considering a team leader with high computational resources. Our experimental testbed also assessed the performance of AinQ on the Zolertia Re-mote Revb and SamL11-xpro boards, which have minimal resources, with results showing that the scheme can be extended to IoT devices with significant resource constraints. We hope to use AinQ as a foundational scheme to build more secure drone-based applications that can be applied to multiple domains in future works. Additionally, we plan to investigate how to accommodate edge drones off-line during the initial group key broadcast phase using either self-healing, mutual healing, or any lightweight technique that would compliment AinQ efficiently.

References

1. Logan Kugler. Real-world applications for drones. *Communications of the ACM*, 62(11):19–21, 2019.
2. Riham Altawy and Amr M. Youssef. Security, privacy, and safety aspects of civilian drones. *ACM Transactions on Cyber-Physical Systems*, 1(2):1–25, 2017.
3. Raja Naeem Akram, Konstantinos Markantonakis, Keith Mayes, Oussama Habachi, Damien Sauveron, Andreas Steyven, and Serge Chaumette. Security, privacy and safety evaluation of dynamic and static fleets of drones. *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017.
4. Eugene Frimpong, Alexandros Bakas, Hai-Van Dang, and Antonis Michalas. Do not tell me what i cannot do! (the constrained device shouted under the cover of the fog): Implementing symmetric searchable encryption on constrained devices. *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security*, 2020.
5. Eugene Frimpong and Antonis Michalas. Iot-cryptodiet: Implementing a lightweight cryptographic library based on ecdh and ecdsa for the development of secure and privacy-preserving protocols in contiki-ng. *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security*, 2020.
6. Benjamin Semal, Konstantinos Markantonakis, and Raja Naeem Akram. A certificateless group authenticated key agreement protocol for secure communication in untrusted uav networks. *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, 2018.
7. Haiyan Sun, Qiaoyan Wen, Hua Zhang, and Zhengping Jin. A novel pairing-free certificateless authenticated key agreement protocol with provable security. *Frontiers of Computer Science*, 7(4):544–557, 2013.
8. Guomin Yang and Chik-How Tan. Strongly secure certificateless key exchange without pairing. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*, 2011.
9. Hu Xiong, Yan Wu, and Zhenyu Lu. A survey of group key agreement protocols with constant rounds. *ACM Computing Surveys*, 52(3):1–32, 2019.
10. Biming Tian, Song Han, Jiankun Hu, and Tharam Dillon. A mutual-healing key distribution scheme in wireless sensor networks. *Journal of Network and Computer Applications*, 34(1):80–88, 2011.
11. Vinod Kumar, Rajendra Kumar, and S.K. Pandey. A computationally efficient centralized group key distribution protocol for secure multicast communications

- based upon rsa public key cryptosystem. *Journal of King Saud University - Computer and Information Sciences*, 32(9):1081–1094, 2020.
12. Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. *Advances in Cryptology - CRYPTO 2003*, page 110–125, 2003.
 13. Sattam S. Al-Riyami and Kenneth G. Paterson. Certificateless public key cryptography. *Advances in Cryptology - ASIACRYPT 2003*, page 452–473, 2003.
 14. Eun-Jung Lee, Sang-Eon Lee, and Kee-Young Yoo. A certificateless authenticated group key agreement protocol providing forward secrecy. *2008 International Symposium on Ubiquitous Multimedia Computing*, 2008.
 15. Pietro Tedeschi, Savio Sciancalepore, Areej Eliyan, and Roberto Di Pietro. Like: Lightweight certificateless key agreement for secure iot communications. *IEEE Internet of Things Journal*, 7(1):621–638, 2020.
 16. Jongho Won, Seung-Hyun Seo, and Elisa Bertino. A secure communication protocol for drones and smart objects. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.
 17. Colin Boyd and Juan Manuel Nieto. Round-optimal contributory conference key agreement. *Public Key Cryptography — PKC 2003*, page 161–174, 2002.
 18. Emmanuel Bresson and Dario Catalano. Constant round authenticated group key agreement via distributed computation. *Public Key Cryptography – PKC 2004*, page 115–129, 2004.
 19. Ratna Dutta and Rana Barua. Constant round dynamic group key agreement. *Lecture Notes in Computer Science*, page 74–88, 2005.
 20. Junghyun Nam, Jinwoo Lee, Seungjoo Kim, and Dongho Won. Ddh-based group key agreement in a mobile environment. *Journal of Systems and Software*, 78(1):73–83, 2005.
 21. Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys*, 35(3):309–329, 2003.
 22. Xinghua Li, Yunwei Wang, Pandi Vijayakumar, Debiao He, Neeraj Kumar, and Jianfeng Ma. Blockchain-based mutual-healing group key distribution scheme in unmanned aerial vehicles ad-hoc network. *IEEE Transactions on Vehicular Technology*, 68(11):11309–11322, 2019.
 23. Sarita Agrawal and Manik Lal Das. Mutual healing enabled group-key distribution protocol in wireless sensor networks. *Computer Communications*, 112:131–140, 2017.
 24. Sarita Agrawal, Jay Patel, and Manik Lal Das. Pairing based mutual healing in wireless sensor networks. *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, 2016.
 25. Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. *Public Key Cryptography*, page 48–63, 2002.
 26. Michael Scott, Kealan McCusker, and Alessandro Budroni. The MIRACL core library. Available at <https://github.com/miracl/core>.
 27. Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas Schmidt. Riot os: Towards an os for the internet of things. *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013.

ASSIGNMENT 2 – (100 MARKS)

Your task is to implement the Symmetric Searchable Encryption scheme illustrated in the paper given below. The goal is to recreate the "Experimental results" section of the paper, using exactly the same dataset.

Do not implement the `SID.Delete` **and** `SID.Modify` **functions**

Note: Implementing the SEV-enabled entities is not required. However, doing so will provide you with a 10% bonus.

Searching in the Dark: A Multi Client Symmetric Searchable Encryption Scheme with Forward Privacy

Alexandros Bakas
Tampere University
alexandros.bakas@tuni.fi

Antonios Michalas
Tampere University
antonios.michalas@tuni.fi

1. Introduction

Symmetric Searchable Encryption (SSE) is a promising encryption technique that allows users to securely outsource private data to an untrusted party – such as a cloud service provider (CSP). More precisely, an SSE scheme allows users to encrypt their data locally with a key that is *not* known to the CSP while at the same time they can search directly over the encrypted data stored in one of the remote locations maintained by the CSP. This, allows them to retrieve files matching a query, without revealing to the CSP (i.e. an untrusted party), neither the data contents, nor the searched keyword. Ideally, an SSE scheme would leak absolutely no information to the CSP during the search process. To achieve this, techniques such as oblivious RAM (ORAM) must be used. However, adopting such techniques will result to a scheme that is even less efficient than downloading the entire encrypted database and perform the search locally [1].

2. Primitives

In this section, we introduce our notation, and we provide a formal definition of a dynamic SSE scheme along with the necessary security definitions.

2.1. Notation

Let \mathcal{X} be a set. We use $x \leftarrow \mathcal{X}$ if x is sampled uniformly from \mathcal{X} and $x \xleftarrow{\$} \mathcal{X}$, if x is chosen uniformly at random. If \mathcal{X} and \mathcal{Y} are two sets, then we denote by $[\mathcal{X}, \mathcal{Y}]$ all the functions from \mathcal{X} to \mathcal{Y} and by $\overline{[\mathcal{X}, \mathcal{Y}]}$ all the injective functions from \mathcal{X} to \mathcal{Y} . $R(\cdot)$ is used for a truly random function, while $R^{-1}(\cdot)$ represents the inverse function of $R(\cdot)$. A function $\text{negl}(\cdot)$ is called negligible, if $\forall n > 0, \exists N_n$ such that $\forall x > N_n: |\text{negl}(x)| < 1/x^n$. If $s(n)$ is a string of length n , we denote by $\overline{s}(l)$ its prefix of length l and by $\underline{s}(l)$, its suffix of length l , where $l < n$. A file collection is represented as $\mathbf{f} = (f_1, \dots, f_z)$ while the corresponding collection of ciphertexts is $\mathbf{c} = (c_{f_1}, \dots, c_{f_z})$. The universe of keywords is denoted by $\mathbf{w} = (w_1, \dots, w_k)$ and the distinct keywords in a file f_i are $w_i = (w_{i_1}, \dots, w_{i_\ell})$. A pseudorandom function is defined as follows:

2.2. Dynamic Symmetric Searchable Encryption

Our construction enables a data owner to share her files with multiple users. However, only the data owner can add and delete files from her collection. We proceed with the definition of a dynamic symmetric searchable encryption scheme.

Definition 1 (DSSE Scheme). *A Dynamic Symmetric Searchable Encryption (DSSE) scheme consists of the following PPT algorithms:*

- $K \leftarrow \text{KeyGen}(1^\lambda)(1^\lambda)$: The data owner generates a secret key K that consists of two keys K_{TA} and K_{SKE} for a IND-CPA secure symmetric key cryptosystem SKE .
- $(\text{In}_{CSP}, \mathbf{c})(\text{In}_{TA}) \leftarrow \text{InGen}(K, \mathbf{f})$: The data owner runs this algorithm to generate the CSP index In_{CSP} and a collection of ciphertexts \mathbf{C} that will be sent to the CSP. Additionally, the index In_{TA} that is stored both locally and in a remote location since it is also outsourced to a trusted authority TA is generated.
- $(\text{In}'_{CSP}, \mathbf{c}')(\text{In}'_{TA}) \leftarrow \text{AddFile}(K, f_i, \text{In}_{TA})(\text{In}_{CSP}, \mathbf{c})$: The data owner is running this algorithm to add a file to her collection of ciphertexts. After a successful run, all indexes and the collection of ciphertexts are updated.
- $(\text{In}'_{CSP}, I_{w_i})(\text{In}'_{TA}) \leftarrow \text{Search}(K, h(w_i), \text{In}_{TA})(\text{In}_{CSP}, \mathbf{C})$: This algorithm is executed by a user in order to search for all files f containing a specific keyword w . After a successful run, the indexes are updated and the CSP also returns to the user a sequence of file identifiers I_w .
- $(\text{In}'_{CSP}, \mathbf{c}')(\text{In}'_{TA}) \leftarrow \text{Delete}(K, c_{id(f_i)}, \text{In}_{TA})(\text{In}_{CSP}, \mathbf{c})$: The data owner runs this algorithm to delete a file from the collection. After a successful run, all indexes are updated accordingly.
- $(\text{In}'_{CSP}, \mathbf{c}')(\text{In}'_{TA}) \leftarrow \text{Modify}(K, c_{id(f_i)}, \text{In}_{TA})(\text{In}_{CSP}, \mathbf{c})$: The data owner runs this algorithm to modify a file that already exists in her collection.

The KeyGen and InGen algorithms do not require any interaction. However, the rest of the algorithms require synchronization between the different entities since In_{TA} is stored both on the owner's side and on the TA .

3. Architecture

In this section, we introduce the system model by describing the entities that participate in our construction.

Users: We denote with $\mathcal{U} = \{u_1, \dots, u_n\}$ the set of all users that have been already registered in a cloud service that allows them to store, retrieve, update, delete and share encrypted files while at the same time being able to search over encrypted data by using our DSSE scheme. The users in our system model are mainly classified into two categories: data owners and simple registered users that have not yet uploaded any data to the CSP. The role of data owner is the most important for our study since it is the one who actually uses the main functions of our scheme to encrypt data locally and create a dictionary that will be sent to the CSP. More precisely, a data owner first needs to locally parse all the data that wishes to upload to the CSP. During this process, she generates three different indexes:

- 1) No.Files[w] which contains a hash of each keyword w along with the number of files that w can be found at
- 2) No.Search[w], which contains the number of times a keyword w has been searched by a user.
- 3) Dict a dictionary that maintains a mapping between keywords and filenames.

Both No.Files[w] and No.Search[w] are of size of $O(m)$, where m is the total number of keywords while the size of Dict is $O(N) = O(nm)$, where n is the total number of files. To achieve the multi-client model, the data owner outsources No.Files[w] and No.Search[w] to a trusted authority TA on the cloud but also keeps a copy locally. These indexes will allow registered users to create consistent search tokens. Dict is finally sent to the CSP.

Cloud Service Provider (CSP): We consider a cloud computing environment similar to the one described in [2]. The CSP must be SEV enabled since core entities will be running in the trusted execution environment offered by SEV-enabled VMs. The CSP storage will consist of the ciphertexts as well as of the dictionary Dict. Each entry of Dict is encrypted under a different symmetric key K_w . Thus, given K_w and the number of files containing a keyword w , the CSP can recover the files containing w .

Trusted Authority (TA): TA is an index storage that stores the No.Files and No.Search indexes that have been generated by the data owner. All registered users can contact the TA to access the No.Files[w] and No.Search[w] values for a keyword w . These values are needed to create the search tokens that will allow users to search directly on the encrypted database. Similarly to the CSP, the TA is also SEV enabled.

SEV: SEV is a security feature that mainly addresses software attacks by isolating VMs from the hypervisor or other VMs on the same platform. It encrypts the VM's memory space with an encryption key that is unique for each VM. The encryption keys are handled by the AMD secure processor which lies on the System on a Chip (SoC). SEV is particularly applicable to cloud services where VMs should not

trust the hypervisor and administrator of their host system. The main advantages of SEV in comparison to its main competitor -Intel SGX- are (1) memory size and (2) efficiency. In particular, SGX allocates only 128MB of memory for software and applications and thus, making it a good candidate for microtransactions and login services. However, SEV's memory is up to the available RAM and hence, making it a perfect fit for securing complex applications. Moreover, in situations where many calls are required, like in the case of a multi-client cloud service, SEV is known to be much faster and efficient than SGX. More information can be found in [3]. The main drawback of SEV is that it is known for not offering memory integrity, which can lead to replay attacks by a malicious hypervisor. However, as of January 2020 AMD has launched an updated version of SEV, called SEV-SNP, which further strengthens VM's isolation with memory integrity [4].

4. Searching in the Dark (SID)

In this section we present our construction. We start by describing Searching in the Dark (SID) – a multi-client DSSE scheme with forward privacy. Then, we present a detailed protocol that shows how SID can be used in a cloud-based service. The description of the protocol that is based on SID is also considered as an important contribution since it allows us to assume a stronger threat model than the one that is assumed in the construction of SID.

Overview. Before we proceed to the formal construction of our scheme, we present a high-level description. The data owner, generates an inverted index Dict, mapping all the keywords to the files containing them. This index will be sent to the CSP along with the ciphertexts. Moreover, the data owner generates two more indexes: (1) No.Files[w] which contain the number of files each keyword appears on, and (2) No.Search[w] which contains the number of times each keyword has been searched for. These indexes are outsourced to the TA where they will be stored but also kept locally on data owner's side. For another user, u_j , to search on the encrypted data for a keyword w_i , she first needs to contact TA to request for the values No.Files[w_i] and No.Search[w_i]. Upon receiving these values, she can generate the search token $\tau_s(w_i)$ and compute the updated values for Dict. Finally, she sends $\tau_s(w_i)$ and the updated dictionary values to the CSP. At the same time, TA will also compute the updated values of Dict and send them to the CSP as well. The CSP will proceed if and only if the values sent from u_j are the same as the ones sent from TA. To delete a file, a data owner must first request the file from the CSP. The CSP will first send the file back to the data owner and then delete all Dict entries that are associated with the file to be deleted. The file is sent back to the data owner so that she will be able to update the No.Files[w] and No.Search[w] indexes that are stored both locally on her machine but also on the TA.

4.1. SID Construction

Our construction constitutes of six different protocols, namely KeyGen, InGen, AddFile, Search, Delete and Modify. Moreover, let $SKE = (Gen, Enc, Dec)$ be a CPA-secure symmetric key encryption scheme and finally, let $h : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ be a cryptographic hash function for the security parameter λ .

Key Generation: The data owner generates the secret key $K = (K_{TA}, K_{SKE})$. K_{TA} is also sent to the TA. This is a probabilistic algorithm run by the data owner.

Algorithm 1 SID.KeyGen

Input: Security parameter λ

Output: $K \leftarrow Gen(1^\lambda)(1^\lambda)$

- 1: $K_{TA} \leftarrow SKE.Gen(1^\lambda)$ \triangleright Common Key of TA and the client
 - 2: $K_{SKE} \leftarrow SKE.Gen(1^\lambda)$
 - 3: return $K = (K_{TA}, K_{SKE})$
 - 4: Send K_{TA} to the TA
-

Indexing: After the data owner generates the secret key K , she runs the InGen algorithm (Algorithm 2) to generate the indexes required by the scheme. In particular, she generates the following three indexes:

- No.Files – stores the total number of files containing a keyword w ,
- No.Search – stores the number of times each keyword has been searched by a user,
- Dict – a mapping of a keyword and the filename that can be found at.

The data owner outsources No.Files and No.Search to the TA but also stores a copy of each index locally. Dict is sent to the CSP. This protocol is treated like a set of AddFile protocols, thus the data owner is required to internally run AddFile (Algorithm 3). Note that upon its generation, Dict is directly sent to the CSP. However, this is not the case for No.Search and No.Files. More precisely, before outsourcing the indexes to the TA, the data owner needs to encrypt them using TA's public key. Although this process is not characterized by its efficiency, it will only occur once and it is a necessary trade-off to achieve a multi-client scheme. Upon reception, TA decrypts the indexes using its private key and stores them in plaintext.

File Insertion: The data owner can add new files to her collection, even after the execution of Algorithm 2. To do so, she retrieves the No.Files[w] and No.Search[w] indexes, that are stored locally on her device. These indexes will allow her to create an add token $\tau_\alpha(f_i)$ for the file f_i that she wishes to add. For each *distinct* keyword $w_{i_j} \in f_i$, she increments No.Files[w_{i_j}] by one and then computes the corresponding address on Dict. Moreover, she computes $c_{f_i} = (K_{SKE}, f_i)$ and she sends the results to the CSP (lines 8-11 of Algorithm 3). As a last step, the data owners sends an acknowledgement to the TA so that TA will also increment No.Files[w_{i_j}] by one.

Search: We now assume that the data owner has successfully shared her secret key K with multiple users

Algorithm 2 SID.InGen

Input: K, f

Output: $(In_{CSP}, c), In_{TA} \leftarrow Index(K, f)$

- 1: $c = \{\}$
 - 2: AllMap = $\{\}$
 - 3: **for all** $f_i \in f$ **do**
 - 4: Run AddFile to generate c_i and Map $_i$ \triangleright
Results are NOT sent to the CSP
 - 5: $c = c \cup c_{f_i}$
 - 6: AllMap = $[\{ AllMap \cup Map_i \}, c_{id(f_i)}]$
 - 7: $In_{TA} = ((No.Files[w], No.Search[w]))$
 - 8: Send (AllMap, c) to the CSP
 - 9: Send In_{TA} to the TA
 - 10: CSP stores AllMap in a dictionary Dict
 - 11: $In_{CSP} = \{Dict\}$
 - 12: $In_{TA} = \{No.Files[w], No.Search[w]\}$
-

Algorithm 3 SID.AddFile

Input: K, f_i, In_{TA}

Output: $(In'_{CSP}, C'), In'_{TA} \leftarrow Add(K, f_i, In_{TA})$
 (In_{CSP}, C')

- 1: Map = $\{\}$
 - 2: **for all** $w_{i_j} \in f_i$ **do**
 - 3: No.Files[w_{i_j}] ++
 - 4: $K_{w_{i_j}} = Enc(K_{TA}, h(w_{i_j}) || No.Search[w_{i_j}])$
 - 5: $addr_{w_{i_j}} = h(K_{w_{i_j}}, No.Files[w_{i_j}] || 0)$ \triangleright
Address in Dict
 - 6: $val_{w_{i_j}} = Enc(K_{SKE}, id(f_i) || No.Files[w_{i_j}])$
 - 7: Map = Map $\cup \{addr_{w_{i_j}}, val_{w_{i_j}}\}$
 - 8: $c_i \leftarrow SKE.Enc(K_{SKE}, f_i)$
 - 9: $\tau_\alpha(f_i) = (c_i, Map)$
 - 10: Send $\tau_\alpha(f_i)$ to the CSP
 - 11: CSP adds c_i into c and Map into Dict
 - 12: Send the updated value of No.Files[w_{i_j}] to TA
 - 13: TA updates No.Files[w_{i_j}]
-

in order for them to access her encrypted data. For a user u_j to create the search token $\tau_s(w_j)$ for a specific keyword w_j , she first needs to request the corresponding values No.Files[w_j] and No.Search[w_j] from the TA (line 1 of Algorithm 4). After u_j receives these values, she can compute the key K_{w_j} for the keyword w_j as $K_{w_j} = Enc(K_{TA}, h(w_j) || No.Search[w_j])$. Apart from that, she increments the No.Search[w_j] value by 1 and computes the updated key for w_j , K'_{w_j} and the new addresses $addr_{w_j}$ on Dict. She will finally store the new addresses in a list L that will be sent to the CSP (lines 3-10 of Algorithm 4). Upon reception, the CSP forwards $(K_{w_j}, No.Files[w_j])$ to the TA to ensure that u_j sent the correct values. At this point, TA will retrieve $h(w_j)$ and No.Search[w_j] by decrypting it, using K_{TA} .

As soon as TA retrieves these values, it can compute $K'[w_j]$ by incrementing No.Search[w_j] by 1. Moreover, it will also compute $addr_{w_j}$. Finally, it will store $addr_{w_j}$ to a list L_{TA} and it will sent it to the CSP (lines 12-19 of Algorithm 4). Upon reception, the CSP will check whether $L_u = L_{TA}$ or not. If $L_u \neq L_{TA}$, the CSP will output \perp and abort the

protocol. If $L_u = L_{TA}$, then the CSP will locate the file identifiers by looking at Dict and replace the corresponding addresses on Dict with the ones received by u_j . The files are then sent back to the user and the CSP sends an acknowledgement to TA in order to increment the value of No.Search[w_j] by one. Finally, this acknowledgement is also forwarded to the data owner, so that she can also update her local indexes.

Algorithm 4 SID.Search

Input: K, w_j, l_{TA}
Output: $(l'_{CSP}, I_{w_j}), l'_{TA} \leftarrow \text{Search}(K, h(w_j), l_{TA})(l_{CSP}, c)$
User:
1: Request the values No.Files[w_j] and No.Search[w_j] for a keyword w_j , from TA
TA:
2: Verifies the user and send back the values
User:
3: $K_{w_j} = \text{Enc}(K_{TA}, h(w_j) || \text{No.Search}[w_j])$
4: $\text{No.Search}[w_j] + +$
5: $K'_{w_j} = \text{Enc}(K_{TA}, h(w_j) || \text{No.Search}[w_j])$
6: $L_u = \{\}$
7: **for** $i = 1$ to $i = \text{No.Files}[w_j]$ **do**
8: $\text{addr}_{w_j} = h(K'_{w_j}, i || 0)$
9: $L_u = L_u \cup \{\text{addr}_{w_j}\}$
10: Send $\tau_{s(w_j)} = (K_{w_j}, \text{No.Files}[w_j], L_u)$ to the CSP
CSP:
11: Forward $(K_{w_j}, \text{No.Files}[w_j])$ to TA
TA:
12: $h(w_j) || \text{No.Search}[w_j] = \text{Dec}(K_{TA}, K_{w_j})$
13: $\text{No.Search}[w_j] = \text{No.Search}[w_j] + 1$
14: $K'_{w_j} = \text{Enc}(K_{TA}, h(w_j) || \text{No.Search}[w_j])$
15: $L_{TA} = \{\}$
16: **for** $i = 1$ to $i = \text{No.Files}[w_j]$ **do**
17: $\text{addr}_{w_j} = h(K'_{w_j}, i || 0)$
18: $L_{TA} = L_{TA} \cup \{\text{addr}_{w_j}\}$
19: Send L_{TA} to the CSP
CSP:
20: **if** $L_u = L_{TA}$ **then**
21: $I_{w_j} = \{\}$
22: **for** $i = 1$ to $i = \text{No.Files}[w_j]$ **do**
23: $c_{id(f_i)} = \text{Dict}[(h(K_{w_j}, i || 0))]$ \triangleright The encryption of the filename
24: $I_{w_j} = I_{w_j} \cup \{c_{id(f_i)}\}$
25: Delete Dict[$(h(K_{w_j}, i || 0))$]
26: Add the new addresses as specified by L_u
27: **else**
28: Output \perp
29: Send I_{w_j} to the user and an acknowledgement to the TA and the Data Owner
TA & Data Owner:
30: No.Search[w_j] + +

File Deletion: Apart from adding files to the her collection, a data owner u_i can also delete files. To do so, she first needs to send a delete token $\tau_d(f_i)$, for the file f_i she wishes to delete, to the CSP. The CSP can identify which entries of Dict correspond to

f_i . However, deleting these entries do not solve the problem since, to delete a file, the indexes No.Files[w] and No.Search[w] need to be updated. To this end, the CSP sends f_i back to u_i before deleting the entries. Upon reception, u_i decrypts the received file, extracts every keyword w_{i_j} contained in f_i and updates the indexes accordingly (lines 2-12 of Algorithm 5). The data owner will then have to compute and send the new addresses and values for Dict to the CSP. Moreover, she sends an acknowledgement to TA, to update its indexes accordingly. Finally, the CSP then proceeds with the deletion of f_i and the update of the database.

Algorithm 5 SID.Delete

Input: $c_{id(f_i)}$ \triangleright Encrypted file name
Output: $(l'_{CSP}, c')(l'_L) \leftarrow \text{Delete}(K, c_{id(f_i)}, l_{CSP}, c)$
Data owner:
1: FileNumber = $\{\}$
2: **for all** $w_{i_j} \in f_i$ **do**
3: **if** No.Files[w_{i_j}] > 1 **then**
4: $\text{addr}_{w_{i_j}} = h(K_{w_{i_j}}, \text{No.files}[w_{i_j}] || 0)$
5: $\text{val}_{w_{i_j}} = \text{Enc}(K_{SKE}, \text{id}(f_i), \text{No.Files}[w_{i_j}])$
6: No.Files[w_{i_j}] – –
7: $\text{naddr} = h(K_{w_{i_j}}, \text{No.files}[w_{i_j}] || 0)$
8: $\text{nval} = \text{Enc}(K_{SKE}, \text{id}(f) || \text{No.Files}[w_{i_j}])$
9: **else**
10: $\text{naddr} = 0^\lambda$
11: $\text{nval} = 0^\lambda$
12: Delete No.Files[w_{i_j}] and No.Search[w_{i_j}]
13: FileNumber = FileNumber $\cup \{h(w_{i_j}), \text{No.Files}[w_{i_j}]\}$
14: Send FileNumber to the TA
15: $\tau_d(f) = \left[\{(\text{addr}_{w_{i_j}}, \text{naddr}_{w_{i_j}}), (\text{val}_{w_{i_j}}, \text{nval}_{w_{i_j}})\}_{i=1}^{\#w \in f_i} \right]$
16: Send $\tau_d(f_i)$ to the CSP
TA:
17: **for all** $h(w_{i_j}) \in \text{FileNumber}$ **do**
18: **if** No.Files[w_{i_j}] > 1 **then**
19: No.Files[w_{i_j}] – –
20: **else**
21: Delete No.Files[w_{i_j}] and No.Search[w_{i_j}]
CSP:
22: **for** $j = 1$ to $j = \#w_i \in f_i$ **do**
23: **if** $\text{newaddr}_{w_{i_j}} = 0$ **then**
24: Delete $\text{addr}_{w_{i_j}}$ and $\text{val}_{w_{i_j}}$
25: **else**
26: $\text{addr}_{w_{i_j}} = \text{naddr}_{w_{i_j}}$
27: $\text{val}_{w_{i_j}} = \text{nval}_{w_{i_j}}$

File Modification: Finally, SID allows a data owner to modify a file (Algorithm 6). If u_i wishes to modify a file f that is stored online, she first needs to run the Delete algorithm to make sure that each entry associated with f will be deleted and that all indexes will be updated accordingly. Recall that during the delete process, u_i has received a local copy of f that has also decrypt it. As a result, u_i modifies the decrypted f locally and then runs the AddFile algorithm for the updated file.

Algorithm 6 SID.Modify

Input: $c_{id}(f_i)$ \triangleright Encrypted file name
 $(\ln'_{CSP}, C'), \ln'_{TA} \leftarrow \text{Add}(K, f_i, \ln_{TA})(\ln_{CSP}, C')$

Data Owner:

- 1: Run the Delete Algorithm for a file f_i
 - 2: Modify f_i
 - 3: Run the AddFile Algorithm with the modified f_i as input
-

5. Experimental Results

Our experiments mainly focused at analyzing the performance of the SID scheme. To do so, we implemented SID in Python 2.7 using the PyCrypto library [8]. To test the overall performance of the underlying dynamic SSE scheme, we used files of different size and structure. More precisely, we selected random data from the Gutenberg dataset [9]. Our experiments focused on three main aspects: (1) Indexing, (2) Searching for a specific keyword and (3) Deleting a specific file. Additionally, our dictionaries were implemented as tables in a MySQL database. In contrast to other similar works, we did not rely on the use of data structures such as arrays, maps, sets, lists, trees, graphs, etc. and we decided to build a more durable implementation with an actual database that properly represents a persistent storage. While the use of a database system decreases the overall performance of the scheme it is considered as more durable and close to a production level. Conducting our experiments by solely relying on data structures would give us better results. However, this performance would not give us any valuable insights about how the scheme would perform outside of a lab. Hence, we would not be able to argue about the actual practicality of our scheme in a proper cloud-based service. Additionally, storing the database in RAM raises several concerns. For example a power loss or system failure could lead to data loss (because RAM is volatile memory). Further to the above mentioned, since we wanted to evaluate the performance of SID under realistic conditions, we decided to use different machines. To this end, we ran our experiments in the following three different machines:

- Intel Core i7-8700 at 3.20GHz (6 cores), 32GB of RAM running Ubuntu 18.04 Desktop operating system;
- Microsoft Surface Book laptop with a 4.2GHz Intel Core i7 processor (4 cores) and 16GB RAM running Windows 10 64-bit;
- Microsoft Surface Book *tablet mode* with a 1.9GHz Intel Core i7 processor and 16GB RAM running Windows 10 64-bit.

As can be seen, apart from the first test-bed where we used a powerful machine with lot of computational power and resources, the other two machines are considered as commodity machines that a typical user can own (especially the tablet). The reason for measuring the performance of SID on such machines and not only in a powerful desktop – like other

similar works – is that in a practical scenario, the most demanding processes of any SSE scheme (e.g. the creation of the dictionary) would take place on a user's machine. Hence, conducting the experiments only on a powerful machine would result in a set of non-realistic measurements.

Dataset: For the needs of our experiments, we created a dataset containing five different sub-datasets with random text files (i.e. e-books in .txt format) from the Gutenberg dataset. The selected datasets ranged from text files with a total size of 184MB to a set of text files with a total size of 1.7GB. It is important to mention that using text files (i.e. pure text in comparison to other formats such as PDF, word, etc.) resulted in a very large number of extracted keywords – thus creating a dictionary containing more than 12 million distinct keywords (without counting stop words). Furthermore, in our implementation we also incorporated a stop words (such as “the”, “a”, “an”, “in”) removal process. This is a common technique used by search engines where they are programmed to ignore commonly used words both when indexing entries for searching and when retrieving them as the result of a search query. This makes both the searching and indexing more efficient while also reducing the total size of the dictionary. Table 1 shows the five different sub-datasets that we used for our experiments as well as the total number of unique keywords that were extracted from each of the incorporated collections of files.

Is our Dataset Realistic (i.e. big enough)? As can be seen from table 1, our dataset is divided into five different sub-datasets ranging from 1,370,023 to 12,124,904 distinct keywords mainly collected from English books. During a project [10] where researchers from Harvard University and Google in 2010 were looking at words in digitised books, they estimated a total of 1,022,000 words and projected that their number would grow by several thousand each year. Furthermore, it is important to mention that this number also includes different forms of the same word. It also includes many archaic words not used in modern English. In addition to that, in the second edition of the Oxford English dictionary, there are approximately 600,000 word forms defined. Again, this includes many words not in common use any more. As a result, even our smallest sub-dataset is almost double in the size of the Oxford English dictionary as well as slightly larger than the total number of words found in digitized books in 2010. As a result our dataset that contains more 12,000,000 distinct keywords can be considered as realistic and can give us valuable insights about how our scheme would behave in real-life scenarios where users would use a cloud service based on our scheme to store their personal files online in encrypted forms.

Indexing & Encryption: The indexing phase is considered as the setup phase of the SSE scheme. During this phase the following three steps take place: (1) reading plaintext files and generating the dictionary, (2) encrypting the files, and (3) building the encrypted indexes. In our experiments, we measured the total

No of TXT Files	Dataset Size	Unique Keywords
425	184MB	1,370,023
815	357MB	1,999,520
1,694	670MB	2,688,552
1,883	1GB	7,453,612
2,808	1.7GB	12,124,904

TABLE 1: Size of Datasets and Unique Keywords

setup time for each one of the sub-datasets shown in table 1. Each process was run ten times on each machine and the average time for the completion of the entire process on each machine was measured. Figure 1 illustrates the time needed for indexing and encrypting text files ranging from 184MB to 1.7GB that resulted to a set of more than 12 million unique keywords. As can be seen from figure 1, the desktop machine needed the less time to complete the setup phase while the tablet took significantly more time not only compared to the desktop but also in comparison to the time needed by the laptop. However, in all three cases it is evident that the scheme can be considered as practical and can even run in typical users' devices. This is an encouraging result and we hope that will motivate researchers to design and implement even better and more efficient SSE schemes but most importantly we hope that will inspire key industrial players in the field of cloud computing to create and launch modern cloud-services based on the promising concept of Symmetric Searchable Encryption. Table 3 summarizes our measurements from this phase of the experiments. As can be seen, to index and encrypt text files that contained 1,370,023 distinct keywords the average processing time was 8.49m, 22.48m and 68.75m for the desktop, laptop and tablet accordingly while for a set of files that resulted in 12,124,904 distinct keywords the average processing time was 68.44m, 203.28m and 545.28m. Based on the fact that this phase is the most demanding one in an SSE scheme the time needed to index and encrypt such a large number of files is considered as acceptable not only based on the size of the selected dataset but also based on the results of other schemes that do not offer forward privacy [11] as well as on the fact that we ran our experiments on commodity machines and not only on a powerful server.

Apart from the generation of the indexer that contains the unique keywords, the incorporated SSE scheme also creates an indexer that maintains a mapping between a keyword (w) and the filename (id) that w can be found at. The total number of the generated pairs in relation to the size of the underlying datasets is shown in table 2.

Search: In this part of the experiments we measured the time needed to complete a search over encrypted data. In our implementation, the search time is calculated as the sum of the time needed to generate a search token and the time required to find the corresponding matches at the database. It is worth mentioning that the main part of this process will be running on the CSP (i.e. a machine with a large pool

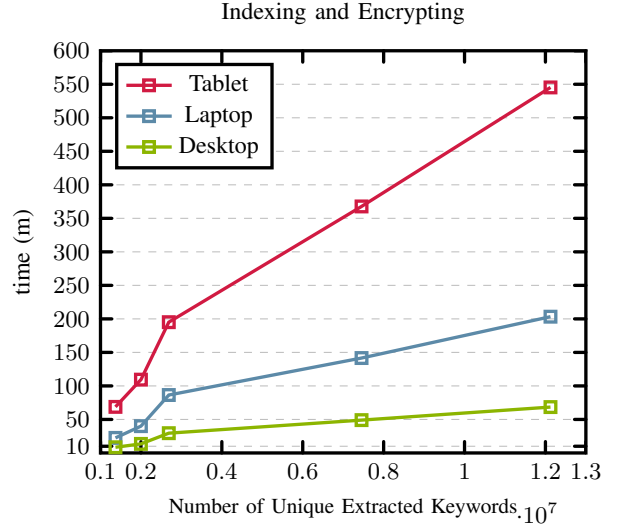


Figure 1: Indexing and Encrypting Files

Unique Keywords	(w, id) Pairs
1,370,023	5,387,216
1,999,520	10,036,252
2,688,552	19,258,625
7,453,612	28,781,567
12,124,904	39,747,904

TABLE 2: Keywords and Filenames pairs

of resources and computational power). To this end, in our experiments we measured the time to generate the search token on the laptop and the tablet (i.e. typical users' machines) while the actual search time was measured using the desktop machine described earlier. On average the time needed to generate the search token on the Surface Book laptop was $9\mu s$ while on the Surface tablet the time for token generation slightly increased to $13\mu s$. Regarding the actual search that is taking place on the CSP it needs to be noted that the actual process is just a series of SELECT and UPDATE queries to the database. More precisely, searching for a specific keyword over a set of 12,124,904 distinct keywords and 39,747,904 addresses required 1.328sec on average while searching for a specific keyword over a set of 1,999,520 distinct keywords and 10,036,252 addresses took 0.131sec.

Delete: To measure the performance of the delete process, we randomly selected 100 different files, performed the delete operation and measured the average processing time for the delete process to be com-

Dataset \ Testbed	Tablet	Laptop	Desktop
184MB	68.75m	22.48m	8.49m
357MB	109.36m	40.00m	13.51m
670	195.09m	86.43m	29.51m
1GB	367.75m	141.60m	48.99m
1.7GB	545.28m	203.28m	68.44m

TABLE 3: Setup time (in minutes) of SID using our dataset and different machines

pleted. We performed the delete queries in our largest dataset containing 12,124,904 distinct keywords and 39,747,904 addresses. The average time to delete a single file and update all the relevant addresses in the database was 1.19min. Even though this time might be considered as high for just deleting a single file it is important to mention that this process will be running on a CSP with a large pool of resources and computational power (e.g. Amazon EC2). Hence, this time is expected to drop significantly on such a computer where more cores will be also utilized. Furthermore, it is important to mention that to properly test the performance of our delete function we need to conduct further experiments where we will also consider the number of keywords contained in the file to be deleted as well as the number of other files that each keyword can be also found at. This is important because it is expected that these factors will heavily affect the performance of the delete function. We plan to conduct further and more detailed experiments on the delete function in our future works.

References

- [1] M. Naveed, "The fallacy of composition of oblivious ram and searchable encryption.," *IACR Cryptology ePrint Archive*, vol. 2015, p. 668, 2015.
- [2] N. Paladi, C. Gehrman, and A. Michalas, "Providing user security guarantees in public infrastructure clouds," *IEEE Transactions on Cloud Computing*, vol. 5, pp. 405–419, July 2017.
- [3] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel sgx and amd memory encryption technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–8, 2018.
- [4] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and more," tech. rep., 01 2020.
- [5] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security* (R. Sion, R. Curtmola, S. Dietrich, A. Kiayias, J. M. Miret, K. Sako, and F. Sebé, eds.), (Berlin, Heidelberg), pp. 136–149, Springer Berlin Heidelberg, 2010.
- [6] A. Michalas, "The lord of the shares: Combining attribute-based encryption and searchable encryption for flexible data sharing," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19*, (New York, NY, USA), pp. 146–155, ACM, 2019.
- [7] A. Bakas and A. Michalas, "Modern family: A revocable hybrid encryption scheme based on attribute-based encryption, symmetric searchable encryption and sgx." *Cryptology ePrint Archive*, Report 2019/682, 2019. <https://eprint.iacr.org/2019/682>.
- [8] "PyCrypto – the Python cryptography toolkit," 2013.
- [9] "Project gutenber," 1971.
- [10] J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, T. G. B. Team, J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak, and E. L. Aiden, "Quantitative analysis of culture using millions of digitized books," *Science*, vol. 331, no. 6014, pp. 176–182, 2011.
- [11] R. Dowsley, A. Michalas, M. Nagel, and N. Paladi, "A survey on design and implementation of protected searchable data in the cloud," *Computer Science Review*, 2017.