SPOT+: secure and privacy-preserving proximity-tracing protocol with efficient verification over multiple contact information

Souha Masmoudi, Maryline Laurent, Nesrine Kaaniche

SPOT+: Secure and Privacy-preserving Proximity-tracing Protocol with Efficient Verification over Multiple Contact Information

Souha Masmoudi[1,2][0000−0002−7194−8240], Maryline Laurent[1,2][0000−0002−7256−3721], and Nesrine Kaaniche[1,2][0000−0002−1045−6445]

[1] Samovar, Télécom SudParis, Institut Polytechnique de Paris, 91120 Palaiseau, France

[2] Member of the Chair Values and Policies of Personal Information, Institut Mines-Telecom, Paris, France

Abstract. At SECRYPT 2022, Masmoudi *et al.* introduced a group signature scheme that offers an aggregated and batch verification over massive proofs of knowledge, named Sevil. The performance analysis of the proposed scheme demonstrates its efficiency and its applicability to real world applications. In this paper, we introduce Spot+, an extension of Sevil to a concrete use-case referred to as a proximity-tracing protocol. Spot+ is a secure and privacy-preserving proximity-tracing protocol that ensures data consistency and integrity and preserves the privacy of users who share their contact information with people in proximity. Spot+ relies on Sevil to significantly improve the performances of the Spot framework [*IEEE Access Journal*, 10, 3208697, (2022)] while supporting aggregated and batch verifications over contact information belonging to multiple users. In comparison with Spot, Spot+ construction allows to reduce computation complexity by 50% and 99% for verifying data integrity and consistency, respectively, when considering an asymmetric pairing type and a 128-bit security level.

Keywords: Group Signatures · Proof of Knowledge · Batch Verification · Proximity-tracing · Privacy.

1 Introduction

With the world-wide adoption of various contact-tracing protocols, several concerns have been raised regarding their practical effectiveness, namely with the increasing number of reported cases. Spot [14] is one promising solution that permit to detect false injections while preserving users' privacy thanks to the usage of group signatures and non interactive proof of knowledge (PoK). Indeed, group signatures enable any group member, referred to as a signer, to sign a message on behalf of the group, while remaining anonymous. As such, verifiers authenticate the signer as a member of the group, but are not able to identify him. For security reasons, verifiers need to ensure that signers are trustworthy while verifying their signing keys, which compromises signers' privacy. To solve

this dilemma and find the trade-off between security and privacy, group signatures might be built upon proof of knowledge (PoK) schemes. That is, the signer proves to verifiers the ownership of the signing key without revealing it, in an interactive or non-interactive session.

Group signatures have been used in several applications namely electronic voting systems [13], privacy-preserving identity management systems [2, 10, 21], etc. Recently, they have been used to design privacy-preserving proximity-tracing protocols [11, 14]. Indeed, in [11] Liu et al. design a proximity-tracing protocol that relies on zero-knowledge proofs and group signatures in order to preserve users' privacy. Users first generate zero-knowledge proofs over their contact information and send them to the doctor in case of infection. Then, after verifying the proofs, the doctor, being a member of a group, generates a group signature over each valid contact information and publishes it in a bulletin board. As such, other users rely on their secret keys to determine their risk score. Later, in the proposal SPOT [14], authors suggested that contact messages generated by users, in a decentralized manner, are first, subject to a real time verification by a centralized computing server and a generation of a partial signature. Then, their integrity is guaranteed thanks to PoK-based group signatures generated by proxies distributed in different geographical areas (i.e., members of the same group). In case of infection, a health authority is responsible for verifying the validity of partial and group signatures. In both solutions, a separate verification (i.e., including the verification of a group signature) should be performed on every contact information.

Giving consideration to the huge number of contact messages and thus, PoK-based group signatures, there is a crucial need to optimize the verification process by verifying multiple contact messages belonging to the same or different users in a single transaction. To this question, Masmoudi *et al.* proposed, in [15], the first group signature scheme, named SEVIL, that offers an efficient, aggregated and batch verification over multiple proofs of knowledge, in particular Groth-Sahai Non-Interactive Witness-Indistinguishable (NIWI) proof scheme [7]. The proposed group signature scheme enables the signer (i.e., member of the group) to preserve his privacy, through the non-disclosure of signing keys, while the verifier still trusts it. The verifier is also able to perform verification over multiple group signatures at once, resulting in performance improvements of up to 50% compared to the naive verification.

In this paper, we present SPOT+, a secure and privacy-preserving proximity-tracing protocol that offers an efficient, aggregated and batch verification over multiple contact information belonging to the same or different users. SPOT+ supports a decentralized certification and a centralized aggregated and batch verification of contact information. Indeed, as in [14], SPOT+ relies on a hybrid architecture that involves:

1. users who share their Ephemeral Bluetooth IDentifiers (EBID) when being in close proximity and who generate a common contact message,
2. a distributed group of proxies that ensure users' anonymity and contact information integrity in a decentralized manner,

3. a centralized server that ensures the correctness of contact information through a real time verification,
4. a centralized health authority that verifies both the integrity and the correctness of multiple contact information provided by one or several infected users.

SPOT+ is designed to support the verification of multiple users' contact information, through the integration and implementation of SEVIL [15]. Indeed, SPOT+ relies on the SEVIL group signature scheme to improve the performances of the verification of contact information integrity and consistency by the health authority. Thus, the contributions of this paper are summarized as follows:

- we design a proximity-tracing protocol that supports efficient batch verification of the correctness and the integrity of contact information without compromising security and users' privacy.
- we evaluate the performances of SPOT+ batch verification and we compare it to the naive verification of each contact information. The comparison demonstrates a gain of up to 50% for contact information integrity verification and 99% of their correctness verification.

The remainder of this paper is organized as follows. Section 2 describes the preliminaries for this work. Section 3 gives an overview of SPOT+ and Section 4 details its phases and algorithms. A security discussion is provided in Section 5 before evaluating SPOT+ performances in Section 6. Section 7 concludes the paper.

2 Preliminaries

In this section, we first, present the batch verification and describe the SEVIL scheme (cf. Section 2.1). Second, we give a brief state of the art of proximity tracing protocols in Section 2.2. Finally, we summarize the properties and the phases of the SPOT protocol in Section 2.3. More details on SEVIL and SPOT can be found in [15] and [14], respectively.

2.1 Batch Verification Over Massive Proofs of Knowledge

Regarding the increasing need to verify the integrity of data, on one hand, and resource constraints' problems, on the other hand, batch verification over multiple signatures was introduced by Naccache *et. al* [16] for DSA-type signatures. It allows to perform the verification of multiple signatures in a single transaction, thus to reduce the computation overhead. Batch verification has been applied to many types of digital signatures, namely group signatures. For instance, group signature schemes that offer batch verification have been proposed to solve resource constraints' problems for vehicular ad hoc networks [20] and IoT systems [1, 22]. Batch verification schemes [12, 15] have been extended to support identification of invalid signatures following the divide-and-conquer approach [17].

Recently, a new group signature offering batch verification over multiple NIWI proofs, called SEVIL has been proposed by Masmoudi *et. al* [15]. SEVIL allows efficient, aggregated and batch verification over Groth-Sahai NIWI proofs, while maintaining a high level of security and privacy. Indeed, verifiers check that signers are trustful without being able to identify them or link several messages signed by the same signer. SEVIL also supports bad signatures identification through the divide-and-conquer approach. In the following, we give a high level description of the original SEVIL scheme through five main algorithms, referred to as Setup, Join, Sign, Batch_Verify and Agg_Verify defined as follows.

- $\mathsf{Setup}() \rightarrow (\mathtt{sk}_g, \mathtt{vk}_g)$ – run by a group manager to set up the group signature parameters. It returns the secret key $\mathtt{sk}_g$ of the group manager, and the group verification key $\mathtt{vk}_g$ that involves the public key of the group manager $\mathtt{pk}_g$ and a common reference string Σ_{NIWI} of a NIWI proof associated with the public key.
- $\mathsf{Join}(\mathtt{sk}_g) \rightarrow (\mathtt{sk}_s, \mathtt{pk}_s, \sigma_k)$ – performed through an interactive session between a group member (i.e., signer) and the group manager. It takes as input the secret key $\mathtt{sk}_g$ of the group manager. The signer generates his pair of private and public keys $(\mathtt{sk}_s, \mathtt{pk}_s)$, and the group manager certifies the signer's public key $\mathtt{pk}_s$ while computing a signature σ_k.
- $\mathsf{Sign}(\mathtt{vk}_g, \mathtt{sk}_s, \mathtt{pk}_s, \sigma_k, m) \rightarrow (\sigma_m, \Pi)$ – run by the signer. It takes as input the group public parameters $\mathtt{vk}_g$, the signer's pair of keys $(\mathtt{sk}_s, \mathtt{pk}_s)$, the signature σ_k over the public key $\mathtt{pk}_s$ and a message m. This algorithm outputs a signature σ_m over the message m and a NIWI proof Π over the two signatures σ_k and σ_m.
- $\mathsf{Batch\_Verify}(\mathtt{vk}_g, \{m_i, \Pi_i\}_{i=1}^{N}) \rightarrow b$ – performed by any verifier. It takes as input the public parameters $\mathtt{vk}_g$, a list of N messages m_i and the associated proofs Π_i sent by the same or multiple signers. This algorithm returns a bit $b \in \{0, 1\}$ stating whether the list of proofs is valid or not.
- $\mathsf{Agg\_Verify}(\mathtt{vk}_{\mathcal{G}}, m, \Pi) \rightarrow b$ – run by any verifier to identify the invalid signature(s), when the Batch_Verify algorithm returns 0 over a list or a sub-list of messages. Given the public parameters $\mathtt{vk}_g$, a message m and the associated proof Π, from an invalid sub-list, the Agg_Verify algorithm returns a bit $b \in \{0, 1\}$ stating whether the proof is valid or not.

2.2 Privacy-preserving Proximity-tracing Protocols

During the COVID-19 pandemic, several proximity-tracing protocols have been proposed to support centralized [9], decentralized [3, 4, 19, 11, 18] or hybrid [5, 8, 14] architectures. Relying on the Bluetooth technology, they enable users to broadcast contact information when they are in proximity with other people and to receive alerts when they are at risk of infection.
Centralized solutions ensure that users receive only correct alerts. Indeed, a centralized server is responsible for generating contact tokens to users and to verify the ones of infected users. These roles allow it to track users and identify their contact lists, which undermines their privacy.

Decentralized solutions have been developed to solve privacy issues. They enable users to generate their own contact tokens and share them with users in proximity, such that they remain anonymous. However, users are exposed to false positive alerts as decentralized solutions do not provide means to verify the correctness of contact information. Additionally, most of the proposed solutions [3, 4, 19] are vulnerable to replay attacks which impacts the reliability of the proximity-tracing application.
Hybrid architecture based solutions have been proposed to leverage the best of both centralized and decentralized architectures, i.e., ensure both security and users' privacy. They rely on a decentralized generation of contact tokens and a centralized verification of infected users' contact information. Indeed, Castelluccia *et al.* proposed Desire [5], a proximity tracing protocol where two users in proximity relies on the Diffie-Hellman key exchange protocol [6] to generate common contact tokens based on their Ephemeral Bluetooth IDentifiers (EBID). As no control and verification are applied on the generated tokens, users are able to collude and merge their contact lists leading to false positive alerts injection. Furthermore, the server responsible for evaluating users' risk scores, is able to de-anonymize users and link their exposure status and risk requests. In [8], contact tokens are also generated in a decentralized way through an interactive session between two users in proximity. However, during the centralized verification, identities of users being in contact with an infected person are revealed to a central server which enables it to track users. In [14], Masmoudi *et al.* proposed a solution, named Spot, that offers a decentralized generation and certification of users' contact information in order to ensure their integrity and consistency. As a result, malicious users are prevented from injecting false positive alerts. The centralized verification allows to verify the integrity and consistency of infected users' contact information without being able to identify with whom they were contact.

2.3 Secure and Privacy-preserving ProximiTy Protocol (SPOT)

Spot is set upon an hybrid architecture that involves four actors, namely a user ($\mathcal{U}$), a server ($\mathcal{S}$), a proxy ($\mathcal{P}$) belonging to a group of proxies, and a health authority ($\mathcal{HA}$). Spot architecture relies on *(i)* a decentralized proxy-based solution to preserve users' privacy (i.e., anonymity) and ensure the integrity of contact information, and *(ii)* a centralized computing server-based solution to ensure contact information integrity and consistency through real-time verification. In the following, we give a high level description of Spot three phases.

The first phase, called Sys_Init, refers to the initialization of the whole system. It includes the generation of the system global parameters and the keys of $\mathcal{S}$ and $\mathcal{HA}$, the setting up of the group of proxies, the joining of proxies to the group and the registration of users at $\mathcal{HA}$. During users' registration, $\mathcal{HA}$ generates, for each user, a unique identifier which is used to generate $\mathcal{U}$'s pair of keys. The user's unique identifier and public key are stored by $\mathcal{HA}$.

The second phase, called Generation, refers to the generation of contact information when two users are in close proximity. They exchange their EBIDs

in order to compute a common contact message. Each user relays the generated message to the server through the group of the proxies. Indeed, the two users should select two different proxies w.r.t. to a comparison of their EBIDs. The two proxies relay the common contact message to the server. $\mathcal{S}$ performs a real-time verification by checking if he receives the same message from two different proxies. If the verification holds, $\mathcal{S}$ partially signs the message and returns the partial signatures to the two proxies. Each proxy extends the given message with the corresponding user's identifier and signs it on behalf of the group. The resulting message and group signature are sent to the user. They are associated to the common contact message to constitute the contact information stored in the user's contact list for for Δ days.

The last phase, called VERIFICATION, refers to the verification of the integrity and consistency of contact information provided by an infected user. For each contact message, $\mathcal{HA}$ performs two verifications. The first one allows to check the validity of the group signature, while the second one allows to verify that the real-time verification over the message has been performed by $\mathcal{S}$. If both verifications hold, the message is added to a set of verified contact messages of infected persons and shared with other users. Otherwise, the message is rejected. As such, SPOT guarantees that users receive only true positive alerts.

3 SPOT+ Protocol

SPOT+ architecture involves the same entities as SPOT, namely the user $\mathcal{U}$, the server $\mathcal{S}$, the group of proxies $\mathcal{P}$ and the health authority $\mathcal{HA}$, as depicted in Figure 1. It involves three main phases, referred to as SYS_INIT, GENERATION and BATCH_VERIFICATION. For ease of presentation, we only illustrate GENERATION and BATCH_VERIFICATION phases in Figure 1 (i.e., we assume that the group of proxies is set up and that users have been already registered at the health authority). SPOT+ includes thirteen PPT algorithms whose chronological sequence is depicted in Figure 2. Note that, for the sake of clarity, we consider *(i)* only one proxy in the sequence diagram, *(ii)* only one user $\mathcal{U}_A$ for the GENERATION phase , and *(iii)* two infected users $\mathcal{U}_A$ and $\mathcal{U}_B$ for the BATCH_VERIFICATION phase.

3.1 Sys_Init Phase

The SYS_INIT phase consists of setting up and initializing the whole system, relying on the following seven algorithms.

- $\mathsf{Set\_params}(\lambda) \rightarrow pp$ – run by a trusted authority to set up the system public parameters pp relying on the security parameter λ. Without loss of generality, we assume the system public parameters pp are an implicit input to the rest of the algorithms.
- $\mathsf{j\_keygen}() \rightarrow (\mathtt{sk}_j, \mathtt{pk}_j)$ – run by a trusted authority to generate the pair of keys of both $\mathcal{HA}$ and $\mathcal{S}$ denoted by the couple $(\mathtt{sk}_j, \mathtt{pk}_j)$ where $j = \{\mathcal{HA}, \mathcal{S}\}$.

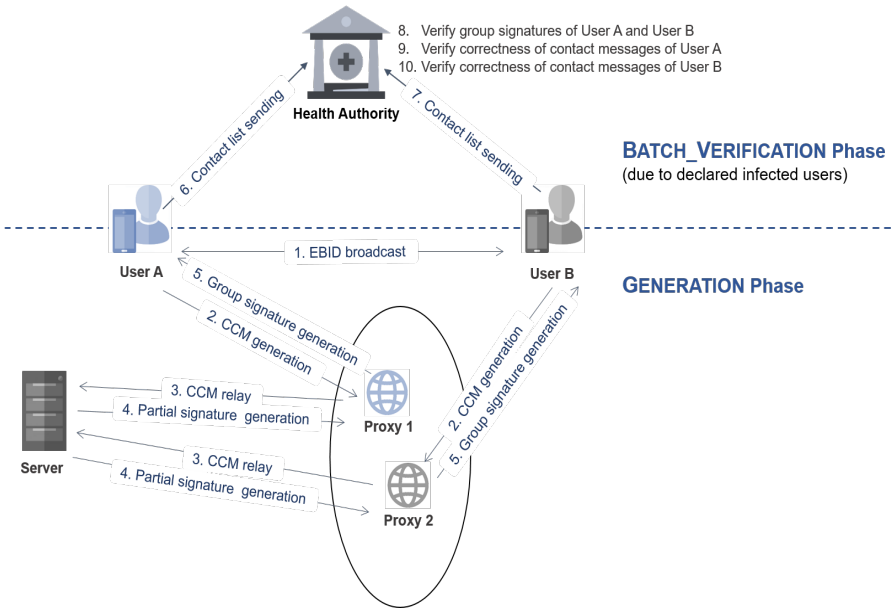


Fig. 1: Overview of the SPOT+ Protocol

- $\mathsf{Setup\_ProxyGr}_{\mathcal{GM}}() \rightarrow (\mathtt{sk}_g, \mathtt{vk}_g)$ – performed by the group manager to define the group of proxies and set up the group signature parameters. These parameters include *(i)* public parameters referred to as the proxies' group verification key $\mathtt{vk}_g$ represented by the couple $(\mathtt{pk}_g, \Sigma_{\mathsf{NIWI}})$ (i.e., $\mathtt{pk}_g$ is the group manager public key and Σ_{NIWI} is the Common Reference String CRS of a Groth-Sahai NIWI proof [7]), and *(ii)* secret parameters namely the secret key $\mathtt{sk}_g$ only known by $\mathcal{GM}$.
- $\mathsf{Join\_ProxyGr}_{\mathcal{P}/\mathcal{GM}}(\mathtt{sk}_g) \rightarrow (\mathtt{sk}_p, \mathtt{pk}_p, \sigma_p)$ – run through an interactive session between $\mathcal{P}$ and $\mathcal{GM}$ to enable a proxy to join the group. For this purpose, $\mathcal{P}$ first generates his pair of keys $(\mathtt{sk}_p, \mathtt{pk}_p)$ and shares the public key $\mathtt{pk}_p$ with $\mathcal{GM}$. $\mathcal{GM}$ generates a signature σ_p over $\mathtt{pk}_p$ relying on the secret key $\mathtt{sk}_g$. The signature σ_p is given back to $\mathcal{P}$. The $\mathsf{Join\_ProxyGr}$ algorithm is performed every time a new proxy joins the group.
- $\mathsf{Set\_UserID}_{\mathcal{HA}}() \rightarrow (t_{\mathcal{U}}, \mathtt{ID}_{\mathcal{U}})$ – performed by $\mathcal{HA}$ when $\mathcal{U}$ installs the proximity-tracing application and asks to be registered. $\mathcal{HA}$ generates a specific secret value $t_{\mathcal{U}}$ and the associated identifier $\mathtt{ID}_{\mathcal{U}}$ for $\mathcal{U}$. Note that $t_{\mathcal{U}}$ is kept secret by $\mathcal{HA}$ and only $\mathtt{ID}_{\mathcal{U}}$ is given back to $\mathcal{U}$.
- $\mathsf{Userkeygen_U}(\mathtt{ID}_{\mathcal{U}}) \rightarrow (\mathtt{sk}_{\mathcal{U}}, \mathtt{pk}_{\mathcal{U}})$ – run by $\mathcal{U}$ to generate his pair of keys $(\mathtt{sk}_{\mathcal{U}}, \mathtt{pk}_{\mathcal{U}})$ relying on his identifier $\mathtt{ID}_{\mathcal{U}}$. Note that the user's public key $\mathtt{pk}_{\mathcal{U}}$ is sent to $\mathcal{HA}$ to be stored in a database DB_{USER}.

Note that the $\mathsf{Set\_UserID}$ and $\mathsf{Userkeygen_U}$ algorithms are performed every time a new user installs the proximity-tracing application.

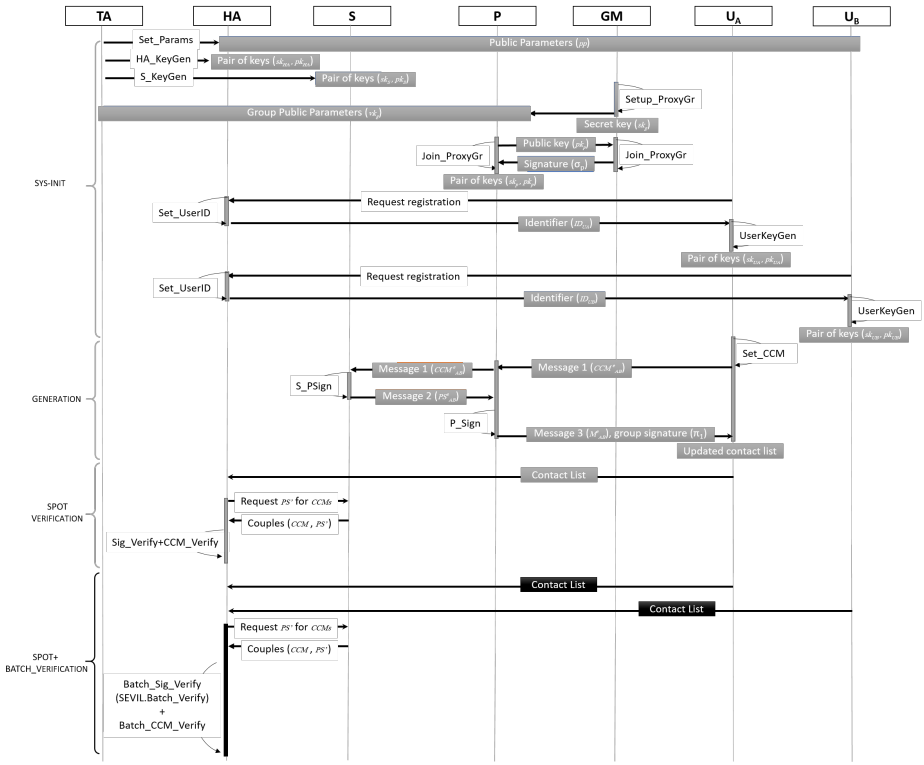


Fig. 2: Workflow of SPOT and SPOT+ Protocols

3.2 Generation Phase

The GENERATION phase occurs when two users $\mathcal{U}_A$ and $\mathcal{U}_B$ are in proximity and they exchange random EBIDs denoted by $\mathtt{D}_A^e$ for $\mathcal{U}_A$ and $\mathtt{D}_B^e$ for $\mathcal{U}_B$. Note that e denotes an epoch in which an EBID remains unchanged. These EBIDs are used to generate a contact message relying on the following algorithms.

- $\mathsf{Set\_CCM}_{\mathcal{U}}(\mathtt{D}_A^e, \mathtt{D}_B^e) \rightarrow \mathtt{CCM}_{AB}^e$ – performed by each of two users $\mathcal{U}_A$ and $\mathcal{U}_B$ being in proximity. Each user generates separately a common contact message $\mathtt{CCM}_{AB}^e$ based on EBIDs $\mathtt{D}_A^e$ and $\mathtt{D}_B^e$. Note that $\mathtt{CCM}_{AB}^e$ is relayed to $\mathcal{S}$ via the group of proxies.
- $\mathsf{S\_PSign}_{\mathcal{S}}(\mathtt{CCM}_{AB}^e, \mathtt{sk}_{\mathcal{S}}) \rightarrow (\mathtt{PS}_{AB}^e, \mathtt{PS'}_{AB}^e)$ – run by $\mathcal{S}$ when receiving two copies of the same contact message from two different proxies $\mathcal{P}_1$ and $\mathcal{P}_2$. $\mathcal{S}$ generates a partial signature represented by the couple $(\mathtt{PS}_{AB}^e, \mathtt{PS'}_{AB}^e)$ in order to be stored with the corresponding common contact message $\mathtt{CCM}_{AB}^e$ for Δ days. $\mathcal{S}$ only returns $\mathtt{PS}_{AB}^e$ to $\mathcal{P}_1$ and $\mathcal{P}_2$.
- $\mathsf{P\_Sign}_{\mathcal{P}_1}(\mathtt{vk}_g, \mathtt{sk}_{p_1}, \mathtt{pk}_{p_1}, \sigma_{p_1}, \mathtt{ID}_{\mathcal{U}_A}, \mathtt{PS}_{AB}^e) \rightarrow (\mathtt{M}_{AB}^e, \sigma_m, \pi)$ – performed by each of the two proxies $\mathcal{P}_1$ and $\mathcal{P}_2$ for the corresponding users $\mathcal{U}_A$ and $\mathcal{U}_B$, respectively. For ease of presentation, we consider only the user $\mathcal{U}_A$ and the

proxy $\mathcal{P}_1$. Relying on the proxies' group public parameters $\mathtt{vk}_g$, his pair of keys $(\mathtt{sk}_{p_1}, \mathtt{pk}_{p_1})$, the signature σ_{p_1}, the identifier $\mathtt{ID}_{\mathcal{U}_A}$ of $\mathcal{U}_A$ and the message $\mathtt{PS}^e_{AB}$, $\mathcal{P}_1$ generates a new message $\mathtt{M}^e_{AB}$, signs it by computing σ_m, and computes a NIWI proof π over the two signatures σ_p and σ_m. $\mathcal{P}_1$ returns the couple $(\mathtt{M}^e_{AB}, \pi)$ to $\mathcal{U}_A$ that stores it along with the common contact message $\mathtt{CCM}^e_{AB}$ in his contact list $CL_{\mathcal{U}_A}$ for Δ days.

3.3 Batch_Verification Phase

The Batch_Verification phase occurs each period of time t after collecting a list N contact messages from infected users. During this phase, $\mathcal{HA}$ verifies the correctness of the N contact messages, in a single transaction. For this purpose, it performs two verifications relying on the following algorithms.

- $\mathsf{Batch\_Sig\_Verify}_{\mathcal{HA}}(\mathtt{vk}_g, \{\mathtt{M}^e_{A_iB_i}, \Pi_i\}_{i=1}^N) \rightarrow b$ – performed by $\mathcal{HA}$ to check, at once, the validity of multiple group signatures (i.e., NIWI proofs) $\{\Pi_i\}_{i=1}^N$ over messages $\{\mathtt{M}^e_{A_iB_i}\}_{i=1}^N$ belonging to different users. Thus, relying on the group public parameters $\mathtt{vk}_g$, the $\mathsf{Batch\_Verify}_{\mathsf{HA}}$ algorithm returns $b \in \{0,1\}$ stating whether the given list of proofs is valid or not.
- $\mathsf{Agg\_Sig\_Verify}_{\mathcal{HA}}(\mathtt{vk}_g, \mathtt{M}^e_{AB}, \Pi) \rightarrow b$ – run by $\mathcal{HA}$ once the $\mathsf{Batch\_Sig\_Verif}_{\mathcal{HA}}$ algorithm returns 0 over a list or a sub-list of contact messages $\mathtt{M}^e_{AB}$ and the corresponding proof Π. Then, relying on the group public parameters $\mathtt{vk}_g$, the $\mathsf{Agg\_Sig\_Verify}$ algorithm is performed over a single message $\mathtt{M}^e_{AB}$ and the corresponding proof Π, from an invalid sub-list. It returns $b \in \{0,1\}$ stating whether the proof is valid or not.
- $\mathsf{Batch\_CCM\_Verify}_{\mathcal{HA}}(\{\mathtt{M}^e_{AB_i}, \mathtt{PS}'^e_{AB_i}\}_{i=1}^N, \mathtt{pk}_{\mathcal{S}}, t_{\mathcal{U}_A}) \rightarrow b$ – performed by $\mathcal{HA}$ for a user $\mathcal{U}_A$ to verify, in a single transaction, that all the contact messages contained in his contact list, have successfully reached $\mathcal{S}$ and been verified in real time. To this end, $\mathcal{HA}$ retrieves from $\mathcal{S}$, the list $\{\mathtt{PS}'^e_{AB_i}\}_{i=1}^N$ w.r.t. $\mathcal{U}_A$'s list of common contact messages $\{\mathtt{CCM}^e_{AB_i}\}_{i=1}^N$. Then, relying on the public key $\mathtt{pk}_{\mathcal{S}}$ of $\mathcal{S}$ and the secret value $t_{\mathcal{U}_A}$ specific to $\mathcal{U}_A$, it returns $b \in \{0,1\}$ stating whether the list of messages $\{\mathtt{CCM}^e_{AB_i}\}_{i=1}^N$ has been correctly generated or not.

4 SPOT+ Algorithms

This section gives a concrete construction of the different phases and algorithms of SPOT+, w.r.t. to the group signature scheme introduced in Section 2.1.

4.1 Sys_Init Phase

- $\mathsf{Set\_params}$ – this algorithm takes as input the security parameter λ and outputs an asymmetric bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3, g_1, g_2, e)$ and a cryptographic hash function $\mathbf{H} : \{0,1\}^* \rightarrow \mathbb{Z}_q$. The system public parameters pp are then represented by the tuple $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3, g_1, g_2, e, \mathbf{H})$.

– $\mathsf{HA\_keygen}$ – this algorithm takes as input the system public parameters pp, selects a random $x \in \mathbb{Z}_q^*$ and outputs the pair of secret and public keys $(\mathtt{sk}_{\mathcal{HA}}, \mathtt{pk}_{\mathcal{HA}})$ of $\mathcal{HA}$ as

$$\mathtt{sk}_{\mathcal{HA}} = x \quad ; \quad \mathtt{pk}_{\mathcal{HA}} = g_2^x$$

– $\mathsf{S\_keygen}$ – this algorithm takes as input the system public parameters pp, selects two randoms $y_1, y_2 \in \mathbb{Z}_q^*$ and generates the pair of secret and public keys $(\mathtt{sk}_{\mathcal{S}}, \mathtt{pk}_{\mathcal{S}})$ of $\mathcal{S}$ as

$$\mathtt{sk}_{\mathcal{S}} = (y_1, y_2) \quad ; \quad \mathtt{pk}_{\mathcal{S}} = (Y_1, Y_2) = (g_2^{y_1}, g_2^{y_2})$$

– $\mathsf{Setup\_ProxyGr}_{\mathcal{GM}}$ – this algorithm takes as input the system public parameters pp and outputs the proxies' group parameters. It is formally defined as follows:

$\underline{\mathsf{Setup\_ProxyGr}_{\mathcal{GM}}(pp):}$
$(\mathtt{sk}_g, \mathtt{vk}_g) \leftarrow \mathsf{Setup}(pp)$, where $\mathtt{vk}_g = (\mathtt{pk}_g, \Sigma_{\mathsf{NIWI}})$
output $(\mathtt{sk}_g, \mathtt{vk}_g)$

– $\mathsf{Join\_ProxyGr}_{\mathcal{P}/\mathcal{GM}}$ – this algorithm takes as input the system public parameters pp and the secret key of the group manager $\mathtt{sk}_g$. It outputs the pair of keys of a proxy group member $(\mathtt{sk}_p, \mathtt{pk}_p)$ and the signature σ_p over the public key $\mathtt{pk}_p$. The $\mathsf{Join\_ProxyGr}$ is formally defined as follows:

$\underline{\mathsf{Join\_ProxyGr}_{\mathcal{P}/\mathcal{GM}}(pp, \mathtt{sk}_g):}$
$(\mathtt{sk}_p, \mathtt{pk}_p, \sigma_p) \leftarrow \mathsf{Join}(pp, \mathtt{sk}_g)$
output $(\mathtt{sk}_p, \mathtt{pk}_p, \sigma_p)$

– $\mathsf{Set\_UserID}_{\mathcal{HA}}$ – this algorithm takes as input the system public parameters pp and selects a secret $t_{\mathcal{U}} \in \mathbb{Z}_q^*$ for a user $\mathcal{U}$. The $\mathsf{Set\_UserID}$ algorithm outputs the couple $(t_{\mathcal{U}}, \mathtt{ID}_{\mathcal{U}})$, where $\mathtt{ID}_{\mathcal{U}}$ is $\mathcal{U}$'s identifier which is computed as follows:

$$\mathtt{ID}_{\mathcal{U}} = h_{\mathcal{U}} = g_2^{t_{\mathcal{U}}}$$

– $\mathsf{Userkeygen}_{\mathcal{U}}$ – this algorithm takes as input the user's identifier $\mathtt{ID}_{\mathcal{U}}$, selects a random $q_{\mathcal{U}} \in \mathbb{Z}_q^*$ and outputs the key pair $(\mathtt{sk}_{\mathcal{U}}, \mathtt{pk}_{\mathcal{U}})$ of $\mathcal{U}$ as:

$$\mathtt{sk}_{\mathcal{U}} = q_{\mathcal{U}} \quad ; \quad \mathtt{pk}_{\mathcal{U}} = {h_{\mathcal{U}}}^{q_{\mathcal{U}}}$$

4.2 Generation Phase

– $\mathsf{Set\_CCM}_{\mathcal{U}}$ – this algorithm takes as input two EBIDs $\mathtt{D}^e_{\mathcal{U}_A}$ and $\mathtt{D}^e_{\mathcal{U}_B}$ belonging to user $\mathcal{U}_A$ and user $\mathcal{U}_B$, respectively, during an epoch e. It returns the corresponding common contact message $\mathtt{CCM}^e_{AB}$ computed as follows:

$$\mathtt{CCM}^e_{AB} = \mathbf{H}(m^e_{AB}) = \mathbf{H}(\mathtt{D}^e_{\mathcal{U}_A} * \mathtt{D}^e_{\mathcal{U}_B})$$

.

– $\mathsf{S\_PSign}_{\mathcal{S}}$ – this algorithm takes as input a common contact message $\mathtt{CCM}^e_{AB}$ and the secret key $\mathtt{sk}_{\mathcal{S}}$ of $\mathcal{S}$, selects a random $r_s \leftarrow \mathbb{Z}_q^*$ and computes the partial signature $(\mathtt{PS}^e_{AB}, \mathtt{PS'}^e_{AB})$ such that:

$$\mathtt{PS}^e_{AB} = \mathtt{CCM}^e_{AB} y_1 r_s + y_2 \quad ; \quad \mathtt{PS'}^e_{AB} = \mathtt{CCM}^e_{AB} r_s$$

– $\mathsf{P\_Sign}_{\mathcal{P}}$ – this algorithm takes as input the proxies' group public parameters $\mathtt{vk}_g$, the secret key $\mathtt{sk}_p$ of $\mathcal{P}$, the signature σ_p over $\mathcal{P}$'s public key, the identifier $\mathtt{ID}_{\mathcal{U}_A}$ of $\mathcal{U}_A$ and the message $\mathtt{PS}^e_{AB}$. It first, computes a new message $\mathtt{M}^e_{AB}$ w.r.t. $\mathtt{ID}_{\mathcal{U}_A}$ and message $\mathtt{PS}^e_{AB}$. It then, generates a signature σ_m over $\mathtt{M}^e_{AB}$ w.r.t. $\mathtt{sk}_p$. Finally, it generates a NIWI proof π over signatures σ_p and σ_m. The $\mathsf{P\_Sign}$ algorithm is formally defined as follows:

$$\begin{array}{l}\mathsf{P\_Sign}_{\mathcal{P}}(\mathtt{vk}_g, \mathtt{sk}_p, \mathtt{pk}_p, \sigma_p, \mathtt{ID}_{\mathcal{U}_A}, \mathtt{PS}^e_{AB}): \\ \hline (\mathtt{M}^e_{AB}, \sigma_m, \pi) \leftarrow \mathsf{Sign}(\mathtt{vk}_g, \mathtt{sk}_p, \mathtt{pk}_p, \sigma_p, \mathtt{ID}_{\mathcal{U}_A}, \mathtt{PS}^e_{AB}) \\ \text{output} \quad (\mathtt{M}^e_{AB}, \sigma_m, \pi)\end{array}$$

4.3 Batch_Verification Phase

– $\mathsf{Batch\_Sig\_Verify}_{\mathcal{HA}}$ – this algorithm takes as input a list of N messages m_i and the corresponding proofs Π_i. Each proof Π_i is composed of six sub-proofs (i.e., two sub-proofs generated over the signature σ_{m_i} w.r.t. the message m_i, and four sub-proofs generated over the signature σ_p w.r.t. the proxy's key $\mathtt{pk}_p$). The list of proofs can be presented as follows:

$$\begin{cases} \{(\vec{\mathcal{A}_{ijm}}, \vec{\mathcal{B}_{ijm}}, \Gamma_{ijm}, t_{ijm})\}_{i,j=1}^{i=N,j=2}, \\ \{(\vec{\mathcal{C}_{ijm}}, \vec{\mathcal{D}_{ijm}}, \pi_{ijm}, \theta_{ijm})\}_{i,j=1}^{i=N,j=2}, \\ \{(\vec{\mathcal{A}_{ilp}}, \vec{\mathcal{B}_{ilp}}, \Gamma_{ilp}, t_{ilp})\}_{i,l=1}^{i=N,l=4}, \\ \{(\vec{\mathcal{C}_{ilp}}, \vec{\mathcal{D}_{ilp}}, \pi_{ilp}, \theta_{ilp})\}_{i,l=1}^{i=N,l=4}. \end{cases}$$

According to the generation of the NIWI proofs over the signatures σ_{m_i} and σ_p , the tuples $\{(\vec{\mathcal{A}_{jm}}, \vec{\mathcal{B}_{jm}}, \Gamma_{jm}, t_{jm})\}_{j=1}^{2}$ and $\{(\vec{\mathcal{A}_{lp}}, \vec{\mathcal{B}_{lp}}, \Gamma_{lp}, t_{lp})\}_{l=1}^{4}$ are unchanged for all N proofs and all proxies. Thus, for a the list of N messages, $\mathsf{Batch\_Sig\_Verify}$ returns $b \in \{0, 1\}$ stating whether the given list of proofs is valid or not, by checking if equations 1 and 2 hold. Note that the $\mathsf{Batch\_Sig\_Verify}_{\mathcal{HA}}$ algorithm is the same as the $\mathsf{Batch\_Verify}$ algorithm.

$$\prod_i \prod_j \left(e(\vec{\mathcal{C}_{ijm}}, \Gamma_m \vec{\mathcal{D}_{ijm}}) \right) = e(\mathtt{U}, \sum_i \sum_j \pi_{ijm}) e(\sum_i \sum_j \theta_{ijm}, \mathtt{V}) \quad (1)$$

$$\prod_l e(\iota_1(\vec{\mathcal{A}_{lp}}), \sum_i \vec{\mathcal{D}_{ilp}}) e(\sum_i \vec{\mathcal{C}_{ilp}}, \iota_2(\vec{\mathcal{B}_{lp}})) \prod_i \prod_l \left(e(\vec{\mathcal{C}_{ilk}}, \Gamma_{lk} \vec{\mathcal{D}_{ilk}}) \right) = \left(\prod_l \iota_3(t_l p)^N \right) e(\mathtt{U}, \sum_i \sum_l \pi_{ilp}) e(\sum_i \sum_l \theta_{ilp}, \mathtt{V}) \quad (2)$$

– $\mathsf{Agg\_Sig\_Verify}_{\mathcal{HA}}$ – this algorithm takes as input a message m belonging to an invalid proof-list and its corresponding proof Π. Using the tuples $\{(\vec{\mathcal{A}_{jm}},\vec{\mathcal{B}_{jm}},\Gamma_{jm},t_{jm})\}_{j=1}^{2}$ and $\{(\vec{\mathcal{A}_{lp}},\vec{\mathcal{B}_{lp}},\Gamma_{lp},t_{lp})\}_{l=1}^{4}$ along with the tuples $\{(\vec{\mathcal{C}_{jm}},\vec{\mathcal{D}_{jm}},\pi_{jm},\theta_{jm})\}_{j=1}^{j=2}$ and $\{(\vec{\mathcal{C}_{lp}},\vec{\mathcal{D}_{lp}},\pi_{lp},\theta_{lp})\}_{l=1}^{l=4}$ derived from Π, the $\mathsf{Agg\_Sig\_Verify}$ outputs $b \in \{0,1\}$ stating whether the proof Π is valid or not, by checking if equations 3 and 4 hold. Note that the $\mathsf{Agg\_Sig\_Verify}_{\mathcal{HA}}$ algorithm is equivalent to the $\mathsf{Agg\_Verify}$ algorithm.

$$\prod_j \left(e(\vec{\mathcal{C}_{jm}},\Gamma_m\vec{\mathcal{D}_{jm}})\right) = e(\mathtt{U},\sum\nolimits_j \pi_{jm})e(\sum\nolimits_j \theta_{jm},\mathtt{V}) \tag{3}$$

$$\prod_l e(\iota_1(\vec{\mathcal{A}_{lp}}),\vec{\mathcal{D}_{lp}})e(\vec{\mathcal{C}_{lp}},\iota_2(\vec{\mathcal{B}_{lp}}))\prod_l \left(e(\vec{\mathcal{C}_{lp}},\Gamma_{lk}\vec{\mathcal{D}_{lp}})\right) = \left(\prod_l \iota_3(t_l p)\right) e(\mathtt{U},\sum\nolimits_l \pi_{lp})e(\sum\nolimits_l \theta_{lp},\mathtt{V}) \tag{4}$$

– $\mathsf{Batch\_CCM\_Verify}_{\mathcal{HA}}$ – this algorithm takes as input the list $\{\mathtt{M}_i\}_{i=1}^{N}$ and the list $\{\mathtt{PS}'_i\}_{i=1}^{N}$ corresponding to the contact messages $\{\mathtt{CCM}_i\}_{i=1}^{N}$ contained in the contact list of user $\mathcal{U}_A$, the server's public key $\mathtt{pk}_{\mathcal{S}}$ and the secret value $t_{\mathcal{U}_A}$ specific to user $\mathcal{U}_A$. The $\mathsf{Batch\_CCM\_Verify}$ algorithm outputs $b \in \{0,1\}$ stating whether the common contact messages have been correctly verified in real time by $\mathcal{S}$ or not, by checking if equation 5 holds:

$$\prod_i \mathtt{M}_i = {Y_1}^{t_{\mathcal{U}_A}\sum_i \mathtt{PS}'_i}{Y_2}^{N t_{\mathcal{U}_A}} \tag{5}$$

5 Security Discussion

This section discusses the security of SPOT+.

Theorem 1. SPOT+ *satisfies unforgeability, unlinkability, anonymity and anti-replay.*

We refer to [14] for formal definitions of the security and privacy properties stated in the theorem. Indeed, unforgeability states that a malicious adversary cannot generate valid contact information without having access to the appropriate keys (i.e., the server and the proxies secret keys). Unlinkability ensures that a curious adversary is not able to link *(i)* two or several common contact messages to the same user during the GENERATION phase and *(ii)* two or several group signatures to the same proxy during the BATCH_VERIFICATION phase. Anonymity means that a curious adversary is not able to identify users involved in a contact list with an infected person. Anti-replay guarantees that a malicious adversary is not able to replay the same contact message in different sessions as a valid contact information. Thus, anti-replay prevents the injection of false positive alerts.

Proof. First, the unforgeability property refers to the unforgeability of both the partial signature generated by the server and the unforgeability of the group

signature generated by a proxy. The unforgeability of the partial signature can be inherited from the unforgeability of SPOT. The unforgeability of the proxies' group signature follows from the unforgeability of the new group signature scheme proposed in [15] proven to be negligible according to the soundness of the Groth-Sahai NIWI proof. Thus, SPOT+ is unforgeable.
Second, for unlinkability, the impossibility to link contact messages issued by the same user follows from the *CCM-unlinkability* property of SPOT. The unlinkability of proxies' group signatures derives from the unlinkability of SEVIL which is proven to be satisfied w.r.t. the computational witness-indistinguishability property of Groth-Sahai NIWI proofs. Thus, SPOT+ is unlinkable.
Third, anonymity follows directly from the anonymity property of SPOT which relies on the impossibility to link common contact messages belonging to the same user.
Finally, for anti-replay, if we suppose that the adversary replays a common contact message `CCM` issued in an epoch e, in another epoch $e' \neq e$, he should be able to produce a new valid partial signature and a corresponding valid group signature over `CCM`, which contradicts the unforgeability property. Thus, SPOT+ ensures the anti-replay.

6 Performance Evaluation

This section discusses the experimental results, presented in Table 1, and demonstrates the performances' improvements introduced by SPOT+. We, first, describe SPOT test-bed in Section 6.1. Then, we analyze, in Section 6.2, the computation performances of SPOT+ w.r.t. the batch verification.

6.1 Test-bed and Methodology

The three phases of SPOT+ including the thirteen algorithms [3] have been implemented and lead to several performance measurements relying on an Ubuntu 18.04.3 machine - with an *Intel* Core $i7@1.30GHz$ processor and $8GB$ memory. This machine runs JAVA version 11, and the associated cryptographic library *JPBC*[4].
The SPOT+ prototype is built with six java classes, namely *TrustedAuthority.java*, *GroupManager.java*, *Proxy.java*, *HealthAuthority.java*, *User.java* and *Server.java*. The *HealthAuthority.java* class encompasses the verification algorithms of both SPOT and SPOT+ (i.e., Sig_Verify, CCM_Verify, Batch_Sig_Verify, Agg_Sig_Verify and Batch_CCM_Verify algorithms).
For the sake of performances' improvement, a multithreading is applied on algorithms P_Sign, Sig_Verify, Batch_Sig_Verify and Agg_Sig_Verify to allow a simultaneous execution of multiple threads. A preprocessing is also applied on algorithms Sig_Verify, Batch_Sig_Verify and Agg_Sig_Verify to prepare in advance

[3] The source code is available at https://github.com/soumasmoudi/SPOTv2
[4] http://gas.dia.unisa.it/projects/jpbc/

variables used several times when running the algorithm.
The implementation tests rely on two types of bilinear pairings, i.e., a symmetric pairing type called *type A* and an asymmetric pairing called *type F*. For each type of pairing, we consider two levels of security, namely 112-bit and 128-bit security levels.
For accurate measurements of the computation time, each algorithm is run 100 times, while considering a standard deviation of an order 10^{-2}. Thus, each experimental result reflects the mean time of 100 tests.

6.2 Computation Overhead

In this section, we focus on the VERIFICATION and BATCH_VERIFICATION phases of SPOT and SPOT+, respectively. We first, discuss the experimental results of both batch and naive verifications, as depicted in Table 1. Then, we give a comparative analysis of the two verifications. Finally, we evaluate the impact of the messages' number on the computation time for a batch verification.

Table 1: Computation Time in milliseconds of SPOT and SPOT+ Verifications

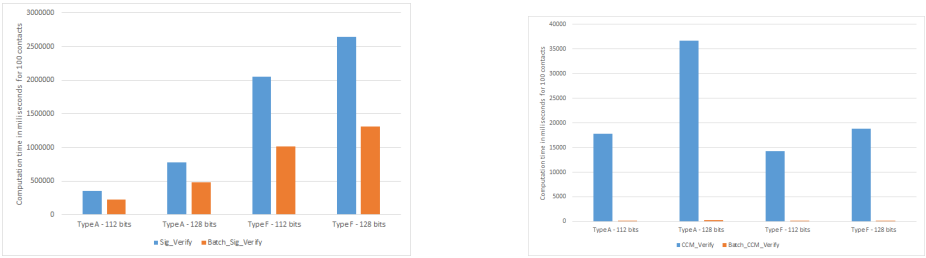
Protocol	Verification Algorithm	Computation time (ms)			
		A/112-bits	A/128-bits	F/112-bits	F/128-bits
SPOT	Sig_Verify [a]	6541	15406	31637	36892
	CCM_Verify [a]	174	360	148	190
SPOT+	Batch_Sig_Verify [b]	222989	485233	1018375	1312879
	Agg_Sig_Verify [a]	3096	6916	16065	18834
	Batch_CCM_Verify [b]	139	281	133	175

NOTE: [a] indicates that the algorithm is performed on a single contact message that is generated by the Set_CCM algorithm; [b] indicates that the algorithm is performed on N messages where $N = 100$ for computation times.

Verification Computation Performances As shown in Table 1, the verification of the correctness of a single contact message, through the Sig_Verify and CCM_Verify algorithms together, requires approximately 7 seconds (resp. 16 seconds) for pairing *type A* and 32 seconds (resp. 37 seconds) for pairing *type F*. Thus, to verify the correctness of 100 contact messages, the SPOT naive verification requires approximately 12 minutes (resp. 27 minutes) pairing *type A* and 53 minutes (resp. more than an hour) for pairing *type F*.
Meanwhile, the Batch_Sig_Verify algorithm that is run to verify 100 messages simultaneously, requires approximately 4 and 8 minutes for pairing *type A* and 17 and 22 minutes for pairing *type F*. However, when it is needed to verify a single message, the Agg_Sig_Verify algorithm requires 3 and 7 seconds for pairing *type A* and 16 and 19 seconds for pairing *type F*. It is worth noticing that, for a number

of messages $N = 100$, the execution of the Batch_Sig_Verify algorithm gives improved computational costs compared to the Agg_Sig_Verify algorithm performed 100 times, separately. Also the Batch_CCM_Verify algorithm gives promising results when verifying 100 messages at once.

Benefit of SPOT+ Batch Verification over SPOT Naive Verification We consider 100 contact messages partially and fully signed with the S_PSign and P_Sign algorithms, respectively. The resulting proofs (resp. partial signatures) are given as input to both Sig_Verify and Batch_Sig_Verify algorithms (resp. both CCM_Verify and Batch_CCM_Verify algorithms). The Sig_Verify and CCM_Verify algorithms are executed 100 times as they allow to perform verification over a single contact message, while the Batch_Sig_Verify and Batch_CCM_Verify algorithms perform the verification of all the 100 contact messages at once. Thus, we compare the computation time required by the naive and batch verification when being executed over 100 messages.



(a) Sig_Verify vs Batch_Sig_Verify (b) CCM_Verify vs Batch_CCM_Verify

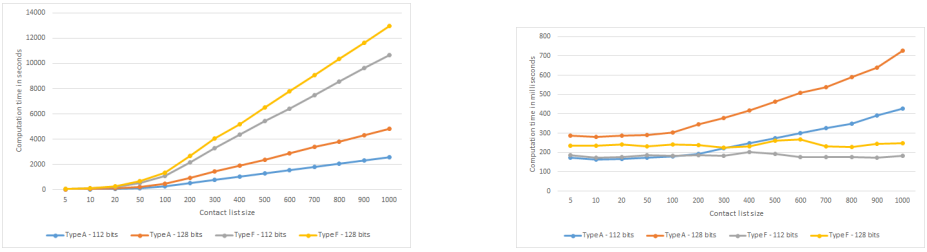
Fig. 3: Computation Time of Batch Verification vs Naive Verification over 100 Messages

Figure 3 confirms that the batch verification is more efficient than the naive one. On the one hand, as depicted in Sub-Figure 3a, the batch verification of proofs reduces the computation time, for verifying 100 messages, by approximately 37%, for pairing *type A* for the two security levels. Indeed, the processing time moves from 356 seconds (resp. 777 seconds) with SPOT naive group signature verification to 223 seconds (resp. 485 seconds) with SPOT+ batch group signature verification. For pairing *type F* and the two security levels, the gain of batch verification reaches 50%, since the processing time is moving from 2048 seconds (resp. 2642 seconds) to 1018 seconds (resp. 1313 seconds).
These results are substantiated by the decrease in the number of pairing functions performed during verification. Indeed, to perform verification over N messages, the Sig_Verify algorithm of SPOT requires $30N$ pairing functions, while the Batch_Sig_Verify algorithm of SPOT+ only requires $6N + 9$ pairing functions. These theoretical results expect that the gain reaches approximately 80%, which

is higher than the gain obtained through experimentation. This difference can be explained by the number of group elements addition operations introduced while aggregating the verification equations (i.e., $14N$ addition operations). With consideration to the *JPBC* library benchmark[5], it is worth noticing that elementary addition operations are more consuming for pairing *type A* than for pairing *type F*. As a result, the gain is more important for pairing *type F*.

On the other hand, Sub-Figure 3b shows that the batch verification over 100 partial signatures, reduces the computation time by 99% for the two types of pairings with the two different levels of security. Indeed, to verify N partial signatures, the CCM_Verify algorithm requires $2N$ exponentiations, while the Batch_CCM_Verify algorithm requires only two exponentiations and N multiplications.

Impact of Contact list Size on the Verification Referring to equations 1, 2 and 5, it is worth mentioning that the computation time of both Batch_Sig_Verify and Batch_CCM_Verify algorithms varies according to the number N of contact messages verified. Indeed, as the number of messages grows, the number of pairing functions, exponentiations and multiplications becomes more important. For this purpose, we evaluate the computation time of the Batch_Sig_Verify and Batch_CCM_Verify algorithms when varying the number of messages from 5 to 1000. Note that all contact messages are partially and fully signed with the S_PSign and P_Sign algorithms, respectively.



(a) Batch_Sig_Verify Algorithm (b) Batch_CCM_Verify Algorithm

Fig. 4: Influence of Contact List Size on Batch Verification Computation Time

Both Figures 4a and 4b show that the computation time of both Batch_Sig_Verify and Batch_CCM_Verify algorithms increases w.r.t the number of messages, for the two types of pairings and the two security levels.

On the one hand, for the Batch_Sig_Verify algorithm, when varying the size of the contact list from 5 to 1000, the computation time varies from 15 to 2602 seconds (resp. from 26 to 4817) for the pairing *type A* 112-bit (resp. pairing *type A* 128-bit). For pairing *type F*, the computation time varies from 59 to

[5] http://gas.dia.unisa.it/projects/jpbc/benchmark.html

10677 seconds (resp. 72 to 12978) for the 112-bit security level (resp. the 128-bit security level).

On the other hand, for the Batch_CCM_Verify algorithm, when the size of the contact list varies from 5 to 1000, the computation time, for the pairing *type A*, varies from 174 to 426 milliseconds, for the 112-bit security level (resp. from 287 to 728, for the 128-bit security level). For the pairing *type F*, the computation time is almost constant. The curve slopes are very low compared to those for pairing *type A*. This can be justified by the fact that (i) we have a constant number of exponentiations and (ii) the elementary multiplication operations are more consuming for the pairing *type A* compared to the pairing *type F*.

7 Conclusion

In this paper, we introduce a concrete construction of an efficient, secure and privacy-preserving proximity-tracing protocol, referred to as SPOT+. The proposed protocol offers a batch verification over multiple contact messages, in an effort to improve verification performances of the SPOT protocol [14] w.r.t. the group signature scheme proposed in [15]. Our contribution is proven to satisfy security and privacy requirements of proximity-tracing protocols, namely unforgeability of contact information, non-injection of false positive alerts, unlinkability of users' contact information and anonymity of users being in contact with infected people. Thanks to the implementation of SPOT and SPOT+ algorithms and the comparison of their verification performances, we show that SPOT+ batch verification achieves a gain of up to 50% for verifying the validity of proxies' group signatures. This gain reaches 99% for the verification of the correctness of contact information belonging to the same user.

References

1. Alamer, A.: An efficient group signcryption scheme supporting batch verification for securing transmitted data in the internet of things. Journal of Ambient Intelligence and Humanized Computing (06 2020). https://doi.org/10.1007/s12652-020-02076-x
2. Camenisch, J., Van Herreweghen, E.: Design and implementation of the idemix anonymous credential system. In: CCS 2002. p. 21–30. Association for Computing Machinery, New York, NY, USA (2002)
3. Carmela, T., Mathias, P., Jean-Pierre, H., Marcel, S., James, L., Edouard, B., Wouter, L., Theresa, S., Apostolos, P., Daniele, A., Ludovic, B., Sylvain, C., Kenneth, P., Srdjan, C., David, B., Jan, B., Dennis, J., Marc, R., Patrick, L., Bart, P., Nigel, S., Aysajan, A., et al.: Decentralized privacy-preserving proximity tracing. https://github.com/DP-3T/documents/blob/master/DP3T%20White%20Paper.pdf (2020)
4. Chan, J., Foster, D., Gollakota, S., Horvitz, E., Jaeger, J., Kakade, S.M., Kohno, T., Langford, J., Larson, J., Singanamalla, S., Sunshine, J., Tessaro, S.: Pact: Privacy-sensitive protocols and mechanisms for mobile contact tracing. arXiv:2004.03544 (2020)

5. Claude, C., Nataliia, B., Antoine, B., Mathieu, C., Cedric, L., Daniel, L.M., Vincent, R.: Desire: A third way for a european exposure notification system. https://github.com/3rd-ways-for-EU-exposure-notification/project-DESIRE/blob/master/DESIRE-specification-EN-v1_0.pdf (2020)
6. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (1976). https://doi.org/10.1109/TIT.1976.1055638
7. Groth, J., Sahai, A.: Efficient non-interactive proof systems for bilinear groups. In: Smart, N. (ed.) Advances in Cryptology – EUROCRYPT 2008. pp. 415–432. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
8. Hoepman, J.H.: Hansel and gretel and the virus: Privacy conscious contact tracing. arXiv preprint arXiv:2101.03241 (2021)
9. Inria, AISEC, F.: Robert: Robust and privacy-preserving proximity tracing. https://github.com/ROBERT-proximity-tracing/documents/blob/master/ROBERT-specification-EN-v1_1.pdf (2020, [Online accessed June 2022])
10. Isshiki, T., Mori, K., Sako, K., Teranishi, I., Yonezawa, S.: Using group signatures for identity management and its implementation. In: DIM '06 (2006)
11. Joseph, K.L., Man Ho, A., Tsz Hon, Y., Cong, Z., Jiawei, W., Amin, S., Xiapu, L., Li, L.: Privacy-preserving covid-19 contact tracing app: A zero-knowledge proof approach. IACR Cryptol. ePrint Arch, 2020 (528) (2020)
12. Kim, K., Yie, I., Lim, S., Nyang, D.: Batch verification and finding invalid signatures in a group signature scheme. International Journal of Network Security **12**, 229–238 (04 2011)
13. Malina, L., Smrz, J., Hajny, J., Vrba, K.: Secure electronic voting based on group signatures. In: 2015 38th International Conference on Telecommunications and Signal Processing (TSP). pp. 6–10 (2015)
14. Masmoudi, S., Kaaniche, N., Laurent, M.: "spot: Secure and privacy-preserving proximity-tracing protocol for e-healthcare systems". In: IEEE Access. vol. 10, pp. 106400–106414 (2022). https://doi.org/10.1109/ACCESS.2022.3208697
15. Masmoudi., S., Laurent., M., Kaaniche., N.: Sevil: Secure and efficient verification over massive proofs of knowledge. In: Proceedings of the 19th International Conference on Security and Cryptography - SECRYPT. pp. 13–24. INSTICC, SciTePress (2022). https://doi.org/10.5220/0011125800003283
16. Naccache, D., M'Raïhi, D., Vaudenay, S., Raphaeli, D.: Can d.s.a. be improved? complexity trade-offs with the digital signature standard. In: Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994. pp. 77–85. Lecture Notes in Computer Science, Springer (1994). https://doi.org/10.1007/BFb0053426
17. Pastuszak, J., Michałek, D., Pieprzyk, J., Seberry, J.: Identification of bad signatures in batches. In: Public Key Cryptography. pp. 28–45 (03 2004). https://doi.org/10.1007/978-3-540-46588-1_3
18. Pietrzak, K.: Delayed authentication: Preventing replay and relay attacks in private contact tracing. In: Progress in Cryptology – INDOCRYPT 2020. pp. 3–15. Springer International Publishing, Cham (2020)
19. Ronald L., R., Jon, C., Ran, C., Kevin, E., Daniel Kahn, G., Yael Tauman, K., Anna, L., Adam, N., Ramesh, R., Adi, S., Emily, S., Israel, S., Michael, S., Vanessa, T., Ari, T., Mayank, V., Marc, V., Daniel, W., John, W., Marc, Z.: The pact protocol specification (2020)

20. Wasef, A., Shen, X.: Efficient group signature scheme supporting batch verification for securing vehicular networks. In: 2010 IEEE International Conference on Communications. pp. 1–5 (2010). https://doi.org/10.1109/ICC.2010.5502136
21. Yue, X., Xu, J., Chen, B., He, Y.: A practical group signatures for providing privacy-preserving authentication with revocation. In: Security and Privacy in New Computing Environments, Second EAI International Conference, SPNCE 2019, Tianjin, China. pp. 226–245 (06 2019)
22. Zhang, A., Zhang, P., Wang, H., Lin, X.: Application-oriented block generation for consortium blockchain-based iot systems with dynamic device management. IEEE Internet of Things Journal **8**(10), 7874–7888 (2021). https://doi.org/10.1109/JIOT.2020.3041163