SAMBA: A Generic Framework for Secure Federated Multi-Armed Bandits

Radu Ciucanu, Pascal Lafourcade, Gaël Marcadet, Marta Soare

Abstract—The multi-armed bandit is a reinforcement learning model where a learning agent repeatedly chooses an action (pull a bandit arm) and the environment responds with a stochastic outcome (reward) coming from an unknown distribution associated with the chosen arm. Bandits have a wide-range of application such as Web recommendation systems. We address the cumulative reward maximization problem in a secure federated learning setting, where multiple data owners keep their data stored locally and collaborate under the coordination of a central orchestration server. We rely on cryptographic schemes and propose SAMBA, a generic framework for Secure federAted Multi-armed BAndits. Each data owner has data associated to a bandit arm and the bandit algorithm has to sequentially select which data owner is solicited at each time step. We instantiate SAMBA for five bandit algorithms. We show that SAMBA returns the same cumulative reward as the non-secure versions of bandit algorithms, while satisfying formally proven security properties. We also show that the overhead due to cryptographic primitives is linear in the size of the input, which is confirmed by our proof-of-concept implementation.

Index Terms—Learning from distributed data, Data security, Multi-armed bandits, Federated learning, Security in machine learning.

1 INTRODUCTION

Federated learning is a machine learning paradigm where multiple data owners collaborate in solving a learning problem, under the coordination of a central orchestration server (see [1] for a survey). Each data owner’s raw data is stored locally and not exchanged or transferred. The development of machine learning algorithms in federated learning settings is a timely topic, which touches several communities: "a longstanding goal pursued by many research communities (including cryptography, databases, and machine learning) is to analyze and learn from data distributed among many owners without exposing that data” [1]. We tackle this goal by relying on cryptographic techniques to develop a secure framework for learning on distributed data.

In particular, we focus on multi-armed bandits, a reinforcement learning model where a learning agent needs to sequentially decide which “arm” to choose among several options (with unknown reward distributions) available in the environment. After each arm selection, the environment responds with a stochastic reward drawn from the reward distribution associated to the chosen arm. To maximize the cumulative reward, the learning agent has to continuously face the so-called exploration-exploitation dilemma and decide whether to explore by choosing arms with more uncertain associated values, or to exploit the information already acquired by choosing the arm with the seemingly largest associated value. Bandits have practical applications such as Web recommender systems, where the arms are

- R. Ciucanu is with INSA Centre Val de Loire & LIFO, France. radu.ciucanu@insa-cvl.fr
- P. Lafourcade is with Univ. Clermont Auvergne & LIMOS, France. pascal.lafourcade@uca.fr
- G. Marcadet is with Univ. Orleans & LIFO/LIMOS, France. gael.marcadet@etu.univ-orleans.fr
- M. Soare is with Univ. Orleans & LIFO, France. marta.soare@univ-orleans.fr

the recommended items and the rewards are given by the user ratings. More specifically, we tackle the problem of secure cumulative reward maximization in federated multi-armed bandits, a problem that to the best of our knowledge has not been previously studied in the literature. Our goal is to propose a generic federated framework that is guaranteed to return exactly the same cumulative reward as standard bandit algorithms [2] Ch 2], while guaranteeing formally proven security properties.

As depicted in Fig. 1, we assume that the data i.e., the reward functions associated to K bandit arms are stored locally by K data owners (DO1, ..., DOK). The data is potentially sensitive, hence it should remain stored locally and cannot be seen in clear by any participant other than its owner (this is why we depict locks near each DOi). As typically done in federated learning, we assume that the learning algorithm is done by some central orchestration server (referred to as server in the sequel). The data customer (DC) sends a budget N to the server and receives the cumulative reward. Moreover, we assume that the participants are honest-but-curious i.e., they correctly do the required com-

Fig. 1. Instantiation of the federated learning paradigm for cumulative reward maximization in multi-armed bandits.
computations, but try to gain as much information as possible based on the data that they see. In particular, we aim at minimizing the data leakage to the server (this is why we also depict a lock near the server) e.g., the server cannot see rewards produced by each data owner.

To motivate our problem setting, we present an example based on federated learning in recommendation systems [3, 4]. In this case, the K data owners are K local stores, each of them being able to recommend items based on potentially sensitive data. Moreover, the data customer is a parent company that displays on its Web site recommended items that can come from any of the K local stores. Given a budget N (i.e., total number of recommended items that can be sequentially displayed by the parent company), the goal of the parent company is to maximize the cumulative reward (i.e., maximize the sum of obtained user ratings on the recommended items). The bandit algorithm has to decide how to sequentially choose the N recommended items, which should come from the K local stores. The aforementioned recommendation systems motivating example can be easily adapted to other classical federated learning applications where security is of paramount importance e.g., commercial, financial, and medical domains [1].

Our goal is to build a generic federated learning framework such that, given some standard bandit algorithm \( A \), we are able to plug \( A \) in our framework and obtain the same cumulative reward as \( A \), while guaranteeing data security. Our goal could be theoretically achieved by relying on a fully homomorphic encryption (FHE) scheme [3], which allows to compute any function directly in the encrypted domain. Indeed, in theory it would suffice that each data owner encrypts its data with a FHE scheme; then, the server would do the computations needed for cumulative reward maximization directly in the encrypted domain. However, it remains an open question how to build a practical FHE system. Although state-of-the-art FHE systems (SEAL [4] and HElib [5]) have done remarkable progress, computations with real numbers are still limited because of the noise needed for FHE multiplications. Moreover, even simple functions such as comparisons needed in all bandit algorithms (e.g., compute an argmax or a probability matching) require complex and time-consuming computations in FHE systems, even for approximate results and even for recent state-of-the-art algorithms [6, 7]. Since FHE systems cannot be currently used off-the-shelf to propose secure federated bandit algorithms, our approach is based on simpler cryptographic schemes, in conjunction with secure multi-party computation.

Summary of Contributions and Paper Organization

In Section 2, we discuss the positioning of our problem setting w.r.t. the related work. In Section 3 we introduce basic notions on bandit algorithms and cryptographic tools.

In Section 4 we present SAMBA, a generic framework for secure cumulative reward maximization for federated bandits. The key ingredients of SAMBA are:

- We distribute the server computations between two nodes: Controller (that sees only encrypted messages and distributes computation tasks among participants) and Comp (whose only goal is to compare numbers obtained after permuting and masking bandit arm scores). This distribution technique allows to perform comparisons, without revealing to the server neither the bandit arm scores nor the arm pulled at some time step.
- We exchange only encrypted messages such that an external network observer cannot learn any input, output, or intermediate data. Moreover, each data owner can see in clear the raw data pertaining to its bandit arm and nothing else. The data owners communicate only with Controller, with messages encrypted with indistinguishable under chosen-plaintext attack (IND-CPA) cryptographic schemes, namely symmetric AES-GCM [8, 9] and asymmetric Paillier [10].

At the end of SAMBA, we compute the cumulative reward by summing up the rewards from each data owner directly in the encrypted domain, by relying on the additive homomorphic property of Paillier. Hence, neither the data owners nor the server nodes can see in clear the cumulative reward: only the data customer that invested a budget for computing the cumulative reward is able to decrypt it.

We instantiate SAMBA to secure five bandit algorithms: \( \varepsilon \)-greedy, UCB, Thompson Sampling, Softmax, and Pursuit. In Section 5 we provide the theoretical analysis of SAMBA. We show that SAMBA achieves the proven correctness, complexity, and security properties when it is instantiated with any bandit algorithm that satisfies the so-called arm score locality property i.e., computing an arm score does not depend on the other arms. In particular, the aforementioned bandit algorithms satisfy the locality property. In a nutshell, we show that SAMBA enjoys the following features:

- **Correctness:** SAMBA returns exactly the same cumulative reward as the standard (non-secure) bandit algorithms because the cryptographic primitives and distribution of tasks do not change the arm selection strategy w.r.t. the standard algorithms.
- **Complexity:** the number of cryptographic primitives is linear in the input: SAMBA uses \( O(NK) \) AES-GCM primitives and \( O(K) \) Paillier primitives. It is a desirable feature that the number of Paillier primitives does not depend on the budget \( N \) because \( N \) is typically larger than the number of arms \( K \), and AES-GCM is much faster than Paillier.
- **Security:** we summarize the security properties in Fig. 2. We give a brief intuition for each participant:

<table>
<thead>
<tr>
<th>Data</th>
<th>Participant DO</th>
<th>DC</th>
<th>Server</th>
<th>Ext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cumulative reward</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum of rewards for DO</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum of rewards for DO</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arm pulled at time step</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reward at time step</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2. Security properties. The ✓ should be read as: the participant can see in clear the concerned piece of data, whereas an empty case means the opposite. The • should be read as: only if DO is pulled at time step \( t \). Ext should be read as: an external network observer having access to all messages exchanged between participants.  

1. https://github.com/Microsoft/SEAL
– DO₁ can see data concerning arm i and nothing else about other arms, nor about the cumulative reward.
– Only DC can see the cumulative reward for which she spends a budget. She can see only this piece of information for which she pays, and nothing else.
– The server nodes (Controller and Comp) and external observers cannot learn any input, output, and intermediate data.

In Section 6, we report on a proof-of-concept empirical evaluation that shows the feasibility and scalability of SAMBA. We present directions of future work in Section 7.

2 RELATED WORK

In Section 2.1, we discuss the positioning of our problem setting w.r.t the federated learning literature. In Section 2.2, we discuss related works on federated and secure bandits.

2.1 Positioning in the Federated Learning Paradigm

We precisely position our problem setting w.r.t. a state-of-the-art federated learning survey [1]. Our framework is built upon the typical federated learning characteristics:

• Data distribution. Data is generated locally and remains decentralized. Each data owner stores its own data and cannot read the data of the other data owners.

• Orchestration. A central orchestration server organizes the learning, but never sees raw data.

Moreover, among the main federated learning settings (cross-silo vs cross-device) [1], our framework pertains to the cross-silo federated learning setting, whose typical characteristics are:

• Distribution scale. There are rather few data owners (less than 100), which can be different organizations e.g., stores, hospitals.

• Data availability and reliability. Each data owner is assumed to be available when it is required to do some computation tasks and there are no machine failures.

• Primary bottleneck. In general, it might be the computation and communication costs. In our framework, both costs have the same asymptotic big-O complexity.

• Addressability. Each data owner has an id that allows the central orchestration server to access it specifically.

• Statefulness. Each data owner is stateful i.e., it maintains local variables throughout the execution of the entire framework.

• Data partition. The partition should be fixed. In our case, we assume feature-partitioned (vertical) data i.e., each data owner has data pertaining to a single bandit arm.

• Incentive mechanisms. There is the need for incentive mechanisms to ensure honest participation of the data owners, since they may also be business competitors e.g., the local stores from our motivating example from the introduction. We assume a monetary incentive derived from data customer’s budget.

As federated learning threat model, we assume that all participants (data owners, data customer, central orchestration server) are honest-but-curious, which means that they can inspect all received messages but cannot tamper the data and computations needed for the learning algorithm. We assume the classical formulation [11] (Chapter 7.5, where honest-but-curious is denoted semi-honest), in particular (i) each node is trusted: it correctly does the required computations, it does not sniff the network and it does not collude with other nodes, and (ii) an external observer has access to all messages exchanged over the network.

To deal with the honest-but-curious model, we rely on two standard cryptographic techniques [1]:

• Secure multi-party computation. The participants collaborate to simulate a fully trusted third party who can: (i) compute a function of inputs provided by all the participants, and (ii) reveal the computed value to a chosen participant (the data customer), with no party learning anything further.

• Homomorphic encryption. As discussed in the introduction, it is not currently possible to efficiently rely on a fully homomorphic encryption scheme. We can nonetheless rely on the partially homomorphic Paillier scheme that is additive homomorphic and allows to sum up the rewards from each data owner directly in the encrypted domain.

2.2 Positioning w.r.t. Federated and Secure Bandits

To the best of our knowledge, our work is the first that relies on cryptographic techniques to provide data security guarantees for federated multi-armed bandit algorithms.

Multi-armed bandits in a federated learning setting is an emerging topic, with very recent works tackling different bandit models: standard stochastic [3], [4], bandits with graph structure [12], and linear bandits [13]. These works rely on differential privacy to protect the data. A differentially private algorithm takes roughly the same computation time as the standard algorithm, but noise is injected in the data, which alters the arm selection strategy and leads to a different output. On the other hand, a cryptography-based approach does not change the arm selection and outputs the same result as the standard algorithm, at the price of an increased computation time due to the use of cryptographic primitives. Although we share the common goal of data protection in federated bandits, the use of different techniques (differential privacy in the related works vs cryptography in our work) leads to complementary systems, whose different architecture and trade-offs are not comparable. Among all aforementioned works, [3], [4] consider the same standard stochastic bandit model as in this paper. We point out an additional difference: [3], [4] consider a horizontal data partition, with M data owners, each of them having partial information on the same K arms, and local empirical arm means are seen by several participants. In contrast, we focus on a vertical data partition (cf. Section 2.1) and our secure framework guarantees that local data maintained by each data owner is hidden from the other participants.

From a different point of view, there exist secure protocols for bandits outsourced to the honest-but-curious cloud, using cryptographic techniques [14]–[16]. The closest to SAMBA is [14] because it also considers the problem of secure cumulative reward maximization for standard stochastic bandits. There are two main differences between [14] and SAMBA. (i) The data distribution assumptions are different: in [14], all data is outsourced to the cloud, whereas SAMBA focuses on a federated learning setting where data is stored locally by each owner and never exchanged. Consequently,
The respective distributed architectures are intrinsically different. (ii) [14] is catered for securing the UCB algorithm, whereas SAMBA is a generic framework where multiple bandit algorithms can be easily plugged in. Among the algorithms supported in SAMBA, we have UCB and similar argmax-based algorithms. Moreover, SAMBA also supports more complex algorithms where arms are pulled based on a probability matching.

3 Preliminaries

We present bandit algorithms in Section 3.1 and cryptographic tools in Section 3.2. Before that, we introduce some useful notation on which we rely throughout the paper:

- By \([x_1, \ldots, x_n]\) we denote the list containing, in order, the elements \(x_1, \ldots, x_n\).
- Given integers \(x\) and \(y\) such that \(x \leq y\), by \([x, y]\) we denote the list \([x, x+1, \ldots, y]\). By \([x]\) we denote the list \([1, \ldots, x]\).
- By \([x]_i\in[n]\) we denote the list \([x_1, \ldots, x_n]\).
- A permutation \(\sigma : L \rightarrow L\) is a function for which every element occurs exactly once as an image value; by \(\sigma^{-1}\) we denote the inverse of \(\sigma\).
- Given a permutation \(\sigma\), by \(\sigma([x_i]_i\in[n])\) we denote the permuted list \([\sigma(x_1), \ldots, \sigma(x_n)]\).

3.1 Bandit Algorithms

The historical motivation [17] behind the multi-armed bandit model concerns the adaptive design of clinical trials. For a given disease, a doctor can choose among \(K\) drugs with probability of success \(\mu_1, \ldots, \mu_K\) unknown at the beginning of the clinical trial. At each step time \(t\), the doctor chooses a drug \(i \in [K]\) for a patient. If the drug heals the patient, we say that drug generates a reward 1; otherwise, we say that the reward is 0. The \(K\) bandit arms model the effectiveness of the \(K\) treatments available in the clinical trial. The assumption is that the rewards observed from each arm \(i\) are independent samples drawn from a Bernoulli distribution associated to arm \(i\). Maximizing the sum of observed rewards means maximizing the number of healed patients from the clinical trial. The design of efficient multi-arm bandit strategies is a dynamic research topic, also motivated by good empirical performance in a wide range of modern applications, from Web advertisement and recommender systems e.g., [18] to game playing e.g., [19].

In this paper, we consider the typical setting of stochastic multi-armed bandits with Bernoulli rewards. Next, we introduce the notation related to bandit algorithms.

A bandit algorithm takes as input the budget \(N\) and the number of arms \(K\), and gives as output the sum of observed rewards for all arms. The unknown environment of the bandit algorithm consists of \(K\) distributions associated to the \(K\) arms. We consider Bernoulli distributions with expected values \(\mu_1, \ldots, \mu_K\) unknown to the learning agent. The agent has access to a reward function \(\text{pull}(\cdot)\) that can be called \(N\) times. For a chosen arm \(i\), a call to the function \(\text{pull}(i)\) randomly returns 0 or 1 according to the associated Bernoulli distribution, i.e., the probability of returning 1 is \(\mu_i\) and the probability of returning 0 is \(1-\mu_i\). The agent sequentially selects the \(N\) arms to be pulled with the goal of maximizing the sum of rewards.

While SAMBA can incorporate several cumulative reward maximization algorithms with Bernoulli rewards (we refer to Section 5.1 for an analysis of the property needed by bandit algorithms to fit in SAMBA), in this paper we illustrate SAMBA using a representative selection of five textbook algorithms [2], [20], [21], which represent a variety of strategies. To minimize redundancy when presenting the aforementioned collection of algorithms, we present what is common to all of them in Fig. 3. In particular, each bandit algorithm needs to store, for each arm \(i \in [K]\), two variables \(s_i\) (sum of rewards) and \(n_i\) (number of pulls), based on which it can compute \(\mu_i = \frac{s_i}{n_i}\) (empirical mean). Each bandit algorithm has its own strategy for choosing \(i_m\) for each time \(t\), that we present in Fig. 4. We stress that at each time \(t\) only one arm is pulled, thus only the corresponding variables \(s_i\) and \(n_i\) will be updated, while the sum of rewards and the number of pulls for all other arms are not affected. To simplify notation, we drop the index indicating the time \(t\) whenever the variables to be updated at time \(t\) are obvious for the context. In the sequel, by (arm) score of a bandit algorithm \(A\) we mean, depending on \(A\), either the argument of the argmax, or the probability needed to compute a probability matching.

3.2 Cryptographic Tools

SAMBA relies on two cryptographic schemes: Paillier and AES-GCM, which are both IND-CPA secure.

Paillier asymmetric encryption. Paillier’s cryptosystem is additive homomorphic [10]. Let \(m_1\) and \(m_2\) be two plaintexts in \(\mathbb{Z}_n\). The product of the two associated ciphertexts with the public key \(pk\), denoted \(c_1 = E_{pk}(m_1)\) and \(c_2 = E_{pk}(m_2)\), is the encryption of the sum of \(m_1\) and \(m_2\). Indeed, we have: \(E_{pk}(m_1) \cdot E_{pk}(m_2) = E_{pk}(m_1 + m_2)\). We also denote by \(D_{sk}(c)\) the decryption of the cipher \(c\) by the secret key \(sk\).

AES-GCM symmetric encryption. AES is a NIST standard [8] for encrypting messages of 128 bits. To encrypt messages larger than 128 bits, we use AES with the GCM mode (Galois Counter Mode) [9] and denote \(c = Enc(m)\) the encryption of \(m\) and \(m = Dec(c)\) the decryption of \(c\) with the same symmetric key shared between the participants.

IND-CPA. Both Paillier and AES-GCM are indistinguishable under chosen-plaintext attack (IND-CPA), which means that they are secure against attackers that have access only to a polynomial number of encrypted messages (and in
We now present the main bricks in the architecture of SAMBA. The chosen design of SAMBA’s architecture is motivated by a trade-off between its genericity (that is, how easily any multi-armed bandit algorithm can be plugged-in SAMBA) and its specificity (that is, how to optimize SAMBA for the functioning of a specific multi-armed bandit algorithm). The architecture of SAMBA presented in Fig. 3 and explained hereafter allows to directly plug-in four of the five bandits algorithms instantiated earlier (UCB, Thompson Sampling, $\varepsilon$-greedy and Softmax). Then, in Section 4.3 we show how a simple modification to the architecture of SAMBA leads to decrease the number of encryptions and communications needed for algorithms where the arm selection is done randomly for some time steps (as is the case, for instance in $\varepsilon$-greedy). Finally, in Section 4.2 we
present the extension needed in SAMBA such that Pursuit can be easily transformed into its federated version, while enjoying the same security guarantees.

4.1.1 Parameter Setup (Fig. 5(a))

At Step 0, DC sends to Controller the budget $N$ and the algorithm $A$ that should be used for cumulative reward maximization. We list in Fig. 4 the algorithms currently supported in SAMBA, hence we allow DC to send to Controller any value from the column “Algorithm $A$” in Fig. 4, i.e., the name of the algorithm + the algorithm-specific parameters, when applicable. Such parameters are $\varepsilon$ (if $\varepsilon$ is fixed) or $f_\varepsilon$ (if $\varepsilon$ is decreasing over time) for $\varepsilon$-greedy, $\tau$ for Softmax, and $\beta$ for Pursuit.

At Step 1, Controller forwards the received input to $DO_i$ and Comp. Additionally, Controller sends to $DO_i$ a randomly generated $seed_{\alpha_i}$, which is used to draw random masks $\alpha_i$ for the arm scores such that Comp cannot see the real arm scores. As detailed later in the sequel, $\alpha_i$ changes at each time step $t \in [K + 1, N]$.

At the end of Step 1, each $DO_i$ pulls its arm once, and initialize its variables $s_i$ and $n_i$.

4.1.2 Core of SAMBA (Fig. 5(b))

The core of SAMBA (Steps 2–5) is executed for each time step $t \in [K + 1, N]$.

We first illustrate the core of SAMBA with the UCB algorithm, for $K = 3$. Assume that we are at $t = 68$, when $s_1 = 24, s_2 = 10, s_3 = 2$, and $n_1 = 33, n_2 = 24, n_3 = 10$. At
Step 2, each data owner DO$_i$ computes $B_i = \frac{n_i}{\tau} + \sqrt{\frac{2 \ln(t)}{n_i \tau}}$, as presented in Fig. 4. Hence, we get $B_1 \approx 1.23, B_2 \approx 1.01$ and $B_3 \approx 1.12$. The random seed $seed_{ai}$ is useful such that all DO$_i$ independently draw the same random mask needed to hide the real arm scores when computing the argmax. Using $seed_{ai}$, we assume we draw $\alpha_{68} = 0.5$, hence each DO$_i$ needs to multiply its $B_i$ with $\alpha_{68}$ before encrypting with Enc and sending to Controller. Hence, Controller receives Enc(0.62) from DO$_1$, Enc(0.5) from DO$_2$, and Enc(0.56) from DO$_3$. This ends the Step 2 in Fig. 5(b).

When receiving the aforementioned messages, Controller is not able to decrypt because it does not know $\tau$. Before forwarding the messages to Comp that is able to decrypt and compare scores, Controller permutes the messages in order to hide from Comp the correspondence between a score and the DO$_i$ that produced it. We stress that the permutation $\sigma_i$, which is pseudo-random, changes at each time step $t$ because we want to avoid that Comp observes patterns where there is an arm much better than the others. By continuing on our example, assume that $\sigma_{68}(1) = 2, \sigma_{68}(2) = 3, \sigma_{68}(3) = 1$. Hence, at Step 3, Controller sends to Comp the list [Enc(0.56), Enc(0.62), Enc(0.5)]. Then, at Step 4, Comp decrypts and computes argmax = 2, hence it sends to Controller the permuted list of pulling bits [Enc(0), Enc(1), Enc(0)]. At Step 5, Controller inverts the permutation and sends Enc(1) to DO$_1$, and Enc(0) to DO$_2$ and DO$_3$. Then, each DO$_i$ can decrypt its pulling bit. In particular, DO$_1$ can decrypt Enc(1), by computing Dec(Enc(1)) = 1. Since the pulling bit is 1, DO$_1$ pulls its arm and gets reward 1 at time step $t = 68$. We update arm variables: $s_1 = 25$ and $n_1 = 34$. No variable is updated for DO$_2$ or DO$_3$, who both had pulling bits 0. For $t = 69$ and until the end of the budget, Steps 2-5 are repeated.

Plugging Thompson Sampling in Samba is done exactly as for UCB, the only change being the arm score function (cf. Fig. 4). As for UCB, no additional parameter is needed for Thompson Sampling.

Plugging $\varepsilon$-greedy in Samba is also immediate. We recall that for $\varepsilon$-greedy the learning agent explores by choosing randomly the arm to pull at time $t$ with probability $\varepsilon$ and exploits by choosing the best arm to pull at time $t$ with probability $1 - \varepsilon$. Next, we also give details of the steps performed in Samba for securing $\varepsilon$-greedy for a time step $t$. We recall that we have as specific parameter $\varepsilon \in [0, 1]$ which gives the probability to explore, which can be either fixed or decreasing according to a time-depending function $f_{\varepsilon}$. We need another random seed, denoted $seed_{vi}$, that should be generated at Step 0 and send by Controller to each DO$_i$ at Step 1. The random seed $seed_{vi}$ is useful such that at each time step all DO$_i$ independently draw the same random number $x \in [0, 1]$, and compare it to $\varepsilon$ in order to decide whether we shall explore or exploit at some time step i.e., if $x \leq \varepsilon$ then explore, else exploit.

For a concrete example, suppose that for $K = 3$ we have $\varepsilon = 0.1$ (thus, a learning agent explores when choosing the arm to pull at time $t$ with probability 0.1 and exploits when choosing the arm to pull at time $t$ with probability 0.9). Using $seed_{vi}$, we assume we draw $x = 0.7$, hence we need to exploit. As shown in Fig. 4 for exploit in $\varepsilon$-greedy we compute the empirical means $\hat{\mu}_i$ for each DO$_i$, then pull the arm corresponding to the largest empirical mean. This is done in Steps 2-5, exactly as exemplified for UCB.

Alternatively, assume that using $seed_i$, we draw $x = 0.04$, hence we need to explore, i.e., randomly choose an arm. To do so, at Step 2, each DO$_i$ encrypts the same number, that we set to 0 without loss of generality. At Step 3, Controller still permutes the list of encrypted masked values received from DO$_i$, before sending it to Comp. At Step 4, Comp computes an argmax on a permuted list containing $K$ identical values, then sends the corresponding pulling bits to each DO$_i$ and the protocol continues as previously until the end of Step 5. Due to the random permutation, all arms have equal chance of being selected for this explore step and the information about the arm that was chosen does not leak.

We continue with an example for Softmax, where the next pulled arm is chosen as a probability matching based on a Boltzmann distribution with temperature parameter $\tau$. As shown in Fig. 4, the probability to pull DO$_i$ is $e^{\hat{\mu}_i/\tau}/\sum_j e^{\hat{\mu}_j/\tau}$. For the computations needed in the arm selection of Softmax, the frontier between local and global computations is less clear. We remind that to easily plug-in an algorithm in Samba, one must be able to compute the score of a DO$_i$ without seeing the variables of the other DO$_{j\neq i}$. The Samba implementation of Softmax requires: (i) at Step 2, each DO$_i$ computes $e^{\hat{\mu}_i/\tau}$ and then applies a mask as explained previously, and (ii) at Step 4, Comp computes the sum and subsequent fractions after receiving all masked probabilities before sending the pulling bits to Controller. Thanks to the proportion-preserving mask, this operation does not modify the computed probabilities. Hence, Comp can recover the same probability matching as in the standard algorithm. All other steps are done as in the previous examples.

For clarity, we also provide a concrete example of the functioning of Softmax within Samba. For simplicity, we do not explicitly write the cryptographic functions in the following example. For every arm $i \in [K]$, we denote the score $e^{\hat{\mu}_i/\tau}$ by $v_i$. We assume a setting with $K = 3$ and $\tau = 0.1$. At an arbitrary time $t = 98$, assume we have $s_1 = 49, s_2 = 9, s_3 = 11$ and $n_1 = 68, n_2 = 24, n_3 = 5$, hence $\hat{\mu}_1 = 49/68 = 0.72, \hat{\mu}_2 = 0.38$ and $\hat{\mu}_3 = 0.2$. At Step 2, the DO$_1$ compute $v_1 = e^{\hat{\mu}_1/\tau} = e^{0.72/0.1} = 1339.43$, $v_2 = 44.7$, and $v_3 = 7.39$, respectively. Each DO$_i$ uses the same mask, say $\alpha_{98} = 0.15$ (coming from seed $seed_{ai}$), and sends to Controller his masked value $\alpha_{98}v_1 = 200.91, \alpha_{98}v_2 = 6.71$, and $\alpha_{98}v_3 = 1.11$, respectively. At Step 3, Controller permutes the list of received values [200.91, 6.71, 1.11] with a randomly drawn $\alpha_{98}$ and sends the result to Comp, assume [1.11, 6.71, 200.91]. At Step 4, Comp divides each received element by $s = 1.11 + 6.71 + 200.91 = 208.73$, which produces the probability matching list [0.01, 0.03, 0.96]. Each value in the aforementioned list is the probability of an arm to be chosen e.g., the last arm has probability 96% to be chosen, and assume that this arm is actually chosen. Hence, Comp sends the list of pulling bits [0, 0, 1] to Controller who, at Step 5, inverts the permutation $\sigma_{98}^{-1}([0, 0, 1]) = [1, 0, 0]$, and sends each pulling bit to the right DO$_i$.

4.1.2.1 Abstracting score computation and arm selection: To implement the four aforementioned algorithms in the Samba generic protocol, we simply need to instan-
tate the two abstract functions \( \text{ComputeScore} \) and \( \text{SelectArm} \) from Fig. 3.

**ComputeScore:** UCB: return \( \hat{\mu}_i + \sqrt{\frac{2 \ln(N)}{n_i}} \)

\( \varepsilon \)-greedy: return \( \hat{\mu}_i \) or return 0 (depends on \( \varepsilon \), seed)

Thompson Sampling: return \( \arg\max_{\epsilon \in [0,1]} \frac{v_{\sigma(i),t}}{\varepsilon_{\sigma(i),t}} \)

**SelectArm:** \( \varepsilon \)-greedy, UCB, Thompson Sampling:

\( \text{return } \arg\max_{\sigma(i) \in [K]} \left(v_{\sigma(i),t}\right) \)

Softmax: return arm \( \sigma(i) \) with probability \( v_{\sigma(i),t} \)

4.1.3 **Cumulative Reward Computation** (Fig. 5(c))

When the budget has been spent (that is, after observing \( N \) rewards), each \( DO_1 \) can compute its final cumulative reward \( s_n \). We must now communicate \( \sum_{s_i} s_i \) to the Data Client (DC) in a secure way. To do so, we use partial homomorphic encryption within the last two steps (Steps 5 and 6) of SAMBA, as depicted in Fig. 5(c).

Namely, at Step 5, each \( DO_1 \) sends \( E(s_i) \) to the Controller, who does not have the private key needed to decrypt it.

At Step 6, Controller computes \( E(s_1) \cdot \ldots \cdot E(s_K) = E(s_1 + \ldots + s_K) \), using the additive homomorphic property, and sends the results to DC, who has the private key needed to decrypt it and thus obtains the cumulative reward.

4.2 **Extension of SAMBA**

**Pursuit** is more challenging because it requires both argmax and probability matching to choose the next arm to be pulled cf. Fig. 4. Indeed, for all algorithms instantiated until now, we had a single iteration over Steps 2-5, where we computed either an argmax (\( \varepsilon \)-greedy, UCB, Thompson Sampling) or a probability matching (Softmax). Consequently, to instantiate **Pursuit** in SAMBA, we need two iterations at each time step: a first iteration to compute the argmax among all empirical means \( \hat{\mu}_i \), and a iteration to compute a probability matching based on \( p_{i,t} \).

To illustrate the SAMBA instantiation of Pursuit, take an example where \( K = 3 \), and after the initialization (i.e., pull each arm once), we have, \( s_1 = 1, s_2 = s_3 = 0, n_1 = n_2 = n_3 = 1 \), and \( p_{1,3} = p_{2,3} = p_{3,3} = 1/3 \). We are at \( t = K + 1 = 4 \) and let \( \beta = 0.1 \). The two iterations work as follows:

1. The first iteration computes probabilities \( p_{i,4} \). Each \( DO_1 \) sends its encrypted masked \( \hat{\mu}_i \) to Controller. Then, Comp computes an argmax exactly as explained in Section 4.1.2. On our example, arm 1 has probability to be pulled: \( p_{1,4} = \frac{1}{3} + 0.1 \cdot (1 - 1/3) = 0.4 \), while the other arms have probabilities to be pulled: \( p_{2,4} = p_{3,4} = 1/3 + 0.1 \cdot (0 - 1/3) = 0.3 \).

2. The second iteration computes the arm to be pulled according to a probability matching. Each \( DO_1 \) sends the encrypted masked probability \( p_{i,4} \) to Controller, which then forwards to Comp. Then, Comp randomly draws an arm according the the probabilities \( p_{i,4} \), exactly as explained in Section 4.1.2 for Softmax.

To generalize the aforementioned ideas to instantiate arbitrary bandit algorithms that need more iterations over Steps 2-5 of SAMBA, the required modifications are:

- In the pseudocode of Controller, Comp, and \( DO_1 \) in Fig. 6, add another loop for \( l \in \left[ nbIteration_A \right] \) just below each for \( t \in \left[ K + 1, N \right] \). The parameter \( nbIteration_A \) is algorithm-dependent and is 2 in the case of **Pursuit**.

- Add parameter \( l \in \left[ nbIteration_A \right] \) to abstract functions \( \text{ComputeScore} \) and \( \text{SelectArm} \), which for **Pursuit** are:

\( \text{ComputeScore}: \)

\( \text{if} \ l = 1 \text{ return } \hat{\mu}_i \text{ else return } p_{i,t} \) cf. Fig. 4

\( \text{SelectArm}: \)

\( \text{if} \ l = 1 \text{ return } \arg\max_{\sigma(i) \in [K]} \left(v_{\sigma(i),t}\right) \text{ else return } \sigma(i) \text{ with probability } v_{\sigma(i),t} \)

- The semantics of the bits sent by Controller in **Pursuit** is different compared to the case of a single iteration. More specifically, only at the last iteration the bits are effectively used to pull an arm, whereas for the iterations \( l \in \left[ nbIteration_A - 1 \right] \) the bits are simply used to compute the score needed for iteration \( l + 1 \).

**TODO:** until here

In **Pursuit**, the returned selection bits \( b_{i,t} \) does not have the same meaning regarding on the considered iteration: In the first iteration (i), \( b_{i,t} \) notifies the arm \( i \) that it is the best arm, needed to update his probability \( p_{i,t} \). In the second iteration (ii), \( b_{i,t} \) notifies the arm \( i \) is the next pulled arm. This behavior does not fit our protocol due to the selection bit handling, which is much complicated than other algorithms, which need a single iteration. Face to this problem, we introduce a new abstract function **HandleSelect**, whose the goal is to handle the selection bit depending on the strategy and the current core of protocol iteration. For all algorithms cited in this paper where \( nbIteration_A = 1 \) and \( b_{i,t} = 1 \), this function is limited to pull a reward and updating the local variables. For **Pursuit**, assuming \( 1 \leq l \leq nbIteration_A \), **HandleSelect** does either the (i) operation in case where the current iteration index \( l \) is 1, or the (ii) operation where \( l \) is 2.

4.3 **Optimization of SAMBA**

**TODO:** Continue pass M from here.

Previously, we presented the fundamental of SAMBA. This base works well for almost all cited algorithm. In some particular cases such as pulling randomly an arm, the next arm is chosen without regarding on scores, for instance with the \( \varepsilon \)-greedy strategy when \( x \leq \varepsilon \) (explore), done in **SAMBA** by returning a constant by each \( DO_i \) used in the permutation created and applied by the Controller.

We propose to optimize this operation by skipping steps 2 and 3. Instead of receiving scores from each \( DO_i \) through Controller, the core of protocol starts at step 4, with Comp which selects randomly an integer \( r_m \in \left[ K \right] \), used next to build the permuted selection bits, sent to the Controller which inverts the permutation and distributes each selection bit to the right data owner. As the random arm selection performed by Comp is not realized on a permuted list, the selected arm index is not permuted. Nevertheless, when Controller computes the invert of permutation \( \sigma_i^{-1} \left( \{ \text{Enc}(b_{i,t}) \} \right)_{i \in \left[ K \right]} \), it implicitly applies the permutation. Hence, thanks to the random selection performed by Comp and the permutation performed by Controller, each arm has a uniform chance to be pulled and Comp does not know which arm would be selected.

This optimization allows avoiding \( K \) encryption and \( K + 1 \) communications which is desirable. Moreover, as is a subpart of **SAMBA**, security properties hold while
SAMBA is assumed secure. Hence, we modify the standard protocol to propose a the optimized random arm pulling, by introducing a new abstract function \textbf{SelectArchitecture}, which at each time step \( t \in [K + 1, N] \), returns either \textbf{Informed} to notify that steps 2-5 must be used, either \textbf{Random} to notify that random arm pulling optimization must be used. Note that we assume that this function is \textit{deterministic} (For a given time step \( t \in [K + 1, N] \), the result must be always the same) in order to avoid communication lock. In the next of the sequel, we denote by Informed (resp. Random) the architecture involved when Informed (resp. Random) is chosen by \textbf{SelectArchitecture}.

5 Theoretical Analysis

In the previous section, we introduced the fundamental of SAMBA, an extension and an optimisation. In this section, we will only consider the base version of SAMBA including the extension, e.g. with \textit{nbiteration} \textsubscript{A} iterations of steps 2-5. In details, we discuss the genericity of this in Section 5.1 by analyzing the properties needed by bandit algorithms to fit in our protocol. Then, we analyze the correctness of SAMBA in Section 5.2, its complexity in Section 5.3, and its security properties in Section 5.4.

5.1 Genericity

Introduced abstract functions detailed in the previous section allows SAMBA to be specialized for multiple multi-armed bandits algorithms while ensuring security guarantees. Indeed, while SAMBA is designed with some abstractions, they are predictable, and the remaining parts of the protocol is exhaustive. In this paper, we focused on several algorithms because they fit with our model while they have their own logic. To determine the reason behind this compatibility, we extract some properties which ensures that if a standard multi-armed bandits \( A \) respects these properties then \( A \) can be simulated by SAMBA.

To verify that an multi-armed bandits algorithm, we first suppose a distributed version of a standard multi-armed bandits algorithm, to which we add constraint. However, it is not easy to come from the standard version to a distributed one. In subsection 5.1.1 we explain necessary modifications to get a distributed algorithm based on a standard. Then, in subsection 5.1.2 we detail all needed properties that ensure that given distributed standard algorithm fit out protocol.

5.1.1 Distribution

In this subsection, we explain how to distribute any standard multi-armed bandits algorithm that fit with our expectations.

Suppose a standard \( K \)-armed bandits algorithm \( A \), designed to work on a single machine. At each time step \( t \in [N] \), the arm \( i_m \), chosen by the \( A \) strategy, is pulled by using the \textit{pull}(\( i_m \)) function. As a \textbf{first step}, we set up \( K \) independent server denoted \( \text{DO}_i \), that can be viewed as a pulling box, whose the goal is limited to compute a reward by calling the \textit{pull}(\( i \)) function. Hence, the probability to get a reward for an arm \( i \), denoted \( \mu_i \), is only known by the \( \text{DO}_i \). In a such context, the initial machine where \( A \) was executed is now called the central server. When the central server needs to pull an arm \( i_m \) instead of calling the \textit{pull}(\( i_m \)), it will send a request to the \( \text{DO}_i \) server which will call \textit{pull}(\( i_m \)) and return the produced reward to the central server. This modification is limited to the \textit{pull}(\( i \)) function, hence all algorithms \( A \) has a representation that fit this description.

At this point, the central server knows everything about arms variable strategy, including all variables related with an arm \( i \), i.e. \( s_i \) and \( n_i \). As a \textbf{second step} e denote by local variables related with an arm \( i \), all variables indexed by \( i \). We move all local variables related with an arm \( i \) to the \( \text{DO}_i \) node. Notice that all standard algorithms \( A \) modified at step 1 always works with step 2. In fact, instead of access directly to variable, the central server will receive missing local variables from all \( \text{DO}_i \) servers. In other words, local variables are still available, but stored on other servers. Hence, at each time that the central server needs a variable, the \( \text{DO}_i \) sent it.

Once a standard algorithm \( A \) is modified by these two steps, each arm \( i \in [K] \) is represented by an independent server called \( \text{DO}_i \). Moreover, each \( \text{DO}_i \) knows his local variables related with the arm \( i \), i.e. \( s_i \) and \( n_i \). In addition, it has access to the \textit{pull}(\( i \)) function which produces no reward (0) or a reward (1), depending on the probability \( \mu_i \). Finally, the central server, at each time step \( t \), pulls the arm \( i_m \) following the \( A \) strategy, based on variables located in all \( \text{DO}_i \).

5.1.2 Properties for Genericity

In this section, we suppose a distributed multi-armed bandits algorithm \( A' \), obtained by applying previous steps on a standard multi-armed bandits algorithm \( A \). Hence, \( A' \) is composed by \( K + 1 \) servers. Note that we could include another denoted \textit{client}, which would provide the budget \( N \) at the beginning, to the central server. However, this change is minor and does not affect our reasoning.

We define the following properties:

- The central server must be composed by a succession of functions \( f_1, f_2, ..., f_{\text{nbiteration}} \textsubscript{A} \) (might a single one) depending on only of the time step \( t \), and \( K \) values computed by each arm \( i \in [K] \) (represented by \( \text{DO}_i \)). Each function selects an arm, thus returns an integer \( i_m \in [K] \). We assume that function \( f_l \) is aware about the \( l \) index.

- The Arm-Score locality property which state that at each time step \( t \in [K + 1, N] \), each \( \text{DO}_i \) is able to compute a value (i.e. score or probability) based only with his local variables.

We note that the succession of functions \( f_1, f_2, ..., f_{\text{nbiteration}} \textsubscript{A} \) can be merged as a single one, which for a given index \( l \), executes the \( f_l \) function. We state that:

\textbf{Theorem 1.} Let \( A \) a standard multi-armed bandits and \( A' \) the distributed version of \( A \), we state that if \( A' \) respecting the properties cited above, then \( A' \) can be simulated by SAMBA.

Due to space reason, the proof of the theorem is available in appendices.

\textbf{TODO:} Discuss which bandits algorithms cannot be currently done with SAMBA.
5.2 Correctness

The correctness property ensures that, for a given multi-armed bandits algorithm \( \mathcal{A} \), the SAMBA version \( \mathcal{A}' \) will return the exact same cumulative reward than \( \mathcal{A} \). To prove that SAMBA respects the correctness property, the goal is to modify the SAMBA version \( \mathcal{A}' \) in order to obtain the standard generic multi-armed bandits presented in Fig. 3. We assume that \( n_{\text{iteration}} = 1 \), which implies that there are only ComputeScore and SelectArm abstract functions. Moreover, we assume that \( \mathcal{A}' \) fitting SAMBA by respecting the aforementioned properties. First, we remove all cryptographic primitives as well as the permutation done by the Controller. At this point, Controller and Comp have the same knowledge, that is why we merge them in a single entity denoted server. As the mask \( \alpha_i \), which allows the server to see in clear scores computed by all DO\(_i\), respects order and proportion preserving properties, formalized and proved in appendix, it can removed without impact on the obtained cumulative reward. Finally, we merge the server and each DO\(_i\) in a single algorithm, and move ComputeScore inside the SelectArm. As we have an undistributed algorithm, all local variables are known by the machine which execute the algorithm. Hence, we obtain the standard generic multi-armed bandits algorithm. Each property and/or security mechanism we removed does not modify the behavior of the algorithm. Thus, SAMBA returns the same cumulative reward than the standard multi-armed bandits algorithm.

5.3 Complexity

In this section, we show that the asymptotic complexity of SAMBA, the number of communications, and the number of cryptographic operations are theoretically equals to \( O(NKn_{\mathcal{A}}) \) and \( O(NK) \) in practice, where \( n_{\mathcal{A}} \) denotes the number of iteration \( n_{\text{iteration}} \), reduced for clarity.

5.3.1 Asymptotic Complexity

We first study the asymptotic complexity, shown theoretically equals to \( O(NKn_{\mathcal{A}}) \) and \( O(NK) \) in practice. We details the complexity for each node, with the following notation:

- \( c \): Asymptotic complexity of ComputeScore
- \( s \): Asymptotic complexity of SelectArm
- \( h \): Asymptotic complexity of HandleSelect

The DC is called at step 0 to give the executed algorithm \( \mathcal{A} \) to the Controller and at step 6 to receive the total cumulative reward \( R \) sent by the Controller. These steps require only one communication, which implies a constant complexity.

Each DO\(_i\) is involved at steps 1, 2, 5 and 6 of SAMBA. Step 1 requires one communication containing the \( \mathcal{A} \) algorithm parameters, sent by the Controller. During steps 2 and 5, executed \( (N - K) \cdot n_{\mathcal{A}} \) times, there is one communication respectively to and from the Controller. More precisely, each DO\(_i\) does computations at step 2 by calling ComputeScore, and at step 5 with HandleSelect. Finally at step 6 which is executed once, one communication is necessary to transmit \( \mathcal{E}(s_i) \) to the Controller. Thus, the complexity of a DO\(_i\) node is equals to \( O(Nn_{\mathcal{A}}(c + h)) \). In practice, both \( c \) and \( h \) are constant, which leads to the complexity \( O(Nn_{\mathcal{A}}) \).

The Controller is implies at every steps. Steps 0 and 6 have a constant complexity. Steps 1 a 5, both executed once, requires \( K \) communications. All other steps are executed \( (N - K) \cdot n_{\mathcal{A}} \) times. In a hand, Steps 3 and 4 consist of a single communication respectively to and from the Comp node which means a complexity equals to \( O(NKn_{\mathcal{A}}) \). On the other hand, steps 2 and 5 consist of exchanges between the Controller and each DO\(_i\), which means a complexity equals to \( O(NKn_{\mathcal{A}}) \). Thus, the Controller has a theoretical complexity equals to \( O(NKn_{\mathcal{A}}) \) and \( O(NK) \) in practice.

At step 2, the Comp node receives a \( K \) sized list of \( v_i \), and returns, at step 3, the permuted selection bits list by calling SelectArm abstract function. These steps are done \( (N - K) \cdot n_{\mathcal{A}} \) times with a constant number of communications at each time step, then the asymptotic complexity is equals to \( O(NKn_{\mathcal{A}}) \). In practice, \( n_{\mathcal{A}} \) is constant and the SelectArm complexity is linear in the number of arm \( K \).

5.3.2 Number of communications

We show that the theoretical number of communications is \( O(NKn_{\mathcal{A}}) \) and \( O(NK) \) in practice. Steps 0, 1, 6 and 7 are executed once, and the highest complexity is \( O(K) \). The core of our protocol including steps 2, 3, 4 and 5 and executed \( (N - K) \cdot n_{\mathcal{A}} \) times. Steps 3 and 4 implies one communication while steps 2 and 5 require \( K \) communications. Thus, the number of communications of our protocol is \( (N - K)n_{\mathcal{A}} \cdot K \in O(NKn_{\mathcal{A}}) \). In practice, \( n_{\text{iteration}} \) is a constant, the complexity down to \( O(NK) \).

5.3.3 Number of cryptographic encryption

Our protocol bases a major part of his security on symmetric and asymmetric encryption. The asymmetric encryption is slower than the security encryption, but allows to perform some operations e.g. addition and multiplication in the cipher space. Hence, we want to reduce at most as possible the usage of asymmetric encryption. The Paillier cipher is used in a \( O(K) \), which depends only on the number of arms, basically lower than the budget \( N \) in our context. On the other side, the symmetric encryption usage follows the complexity \( O(NKn_{\mathcal{A}}) \) either \( O(NK) \) in practice.

5.4 Security

Our protocol ensures that:

- At the exception of DO\(_i\) itself, no one entity including an external observer can predict the partial cumulative reward \( s_i \) for any \( i \in [K] \) at any turn \( t \in [N] \) better than random.
- At the exception of DC after step 6, no one entity including an external observer can predict \( R \) better than random.
- No one entity including an external observer can predict the pulled arm \( i_m \) except itself at turn \( t \) for any \( t \in [N] \).

We omit formal statements, available in appendices.

The permutation \( \sigma \) is updated at each iteration of the core of protocol, as well as the mask \( \alpha \) when used in the Informed architecture. In a such way, even with several iterations, Comp is not able to get data provided by the same DO\(_i\) in distinct iterations, protected with the same
mask, which would have the effect to allow some operations of these data, and even to leak some sensitive data, e.g. $\mu_i/\mu_n = \tilde{\mu}_i$. Moreover, Comp cannot identify the real pulled arm due to the pseudo-random permutation $\sigma_p$, proved secure thanks to the indistinguishability under permutation (IND-PERM) game, which intuitively state that for a given list, it is not possible to retrieve the initial list, without knowing the initial list and the permutation, with a probability better than random.

6 Experiments

In this section, we present a proof-of-concept empirical study of SAMBA. We explain the setting of the experiments in subsection 6.1 and we discuss the results in subsection 6.2.

6.1 Setting

We show that the overhead due to cryptographic primitives is reasonable, hence our protocols are feasible in practice. More precisely, we show the scalability of our protocols with respect to both parameters $N$ and $K$ through an experimental study using MovieLens [23] and Jester [24] datasets, two datasets contain user ratings for movies, respectively for jokes. The use of ratings for testing bandit algorithms is natural, since the movies/jokes in the dataset correspond to the bandits arms, and a user rating given for a particular movie/joke corresponds to the reward that can be obtained when the corresponding arm is chosen. Similarly to [25], we used a threshold to pre-process data such that the ratings received by each movie/joke could be used as rewards coming from Bernoulli distributions: each rating above a threshold was converted into a reward 1 and each rating below the threshold was converted into a reward 0. The budget used in the experiments varies from $N = 50000$ to $N = 100000$, for the five algorithms instantiated in the paper (Fig. 4), and for $K = 1000$ arms.

TODO: motivation + reference [26] pour expliquer pourquoi le privacy est important y compris avec small data

We implemented the algorithms in Python 3. For AES-GCM we used the Cryptography library [27] and keys of 256 bits. For Paillier, we used the phe library [28] in the default configuration with keys of 2048 bits. We did our experiments in a virtual machine running Ubuntu, located in a server including a 24 cores CPU. The choice of parameters for bandit algorithms is based on an optimization study similar to [20]. Namely, for Pursuit we used with $\beta = 0.5$, for $\epsilon$-greedy, we used $\epsilon = 0.005$, respectively a decreasing factor at 0.95, and for Softmax we used $\tau = 0.002$.

All details concerning the implementation are available on a public GitHub repository [3] including our source, the data, the generated results from which we obtained our plots, and scripts that allow to install the needed libraries and reproduce our plots.

6.2 Results

TODO: continue pass M from here As expected, both distributed secured and unsecured of each algorithm produces the same cumulative reward that the standard version. Experimentation show that the execution time is linear in $K$ and $N$, which match with the stated complexity $O(NK)$ in practice. It means that the overhead of crypto in SAMBA is reasonable and does not impact the complexity. In the case of Pursuit which requires two computation rounds $n_A$ instead of one for all other algorithms, the execution time is, as expected, twice more long ($O(2NK)$). The zoom on the execution time by component shows that Comp is the most time-consuming, which is coherent because of it is the node which performs the most computation over a $K$ sized list.

7 Conclusions and Future Work

We tackled the problem of secure cumulative reward maximization in multi-armed bandits in a federated learning setting where, under the orchestration of a central server, each data owner participating at the cumulative reward computation has the guarantee that its raw data is not seen by any other participant. We proposed SAMBA, a generic secure protocol that is able to easily transform multi-armed bandit algorithms in their secure federated version, while yielding the exact same cumulative reward as their standard (non-secure, non-federated) version. To achieve SAMBA’s security properties, we relied on secure multi-party computations and cryptographic schemes under the honest-but-curious threat model. Through a theoretical analysis and proof-of-concept experiments, we showed that the cryptographic overhead implied by SAMBA remains reasonable in practice. As directions for future work, we plan to extend SAMBA such that it provides security guarantees in more complex threat models and for more complex multi-armed bandit frameworks. More in general, using cryptography to ensure data security for machine learning algorithms is a promising, timely direction. We plan to pursue this direction and to design secure protocols useful for other machine learning paradigms and applications.

References

Fig. 7. The first plots show the average sum of rewards over 150 iterations, on MovieLens (first plot) and Jester (second plot). The best arm has mean reward 0.53 for MovieLens, and 0.50 for Jester. Next, we show the average execution time over 20 iterations, on MovieLens (third plot) and Jester (forth plot). In all four plots we consider settings with $K = 10^2$ arms and an increasing budget $5 \cdot 10^4 \leq N \leq 5 \cdot 10^5$. The dotted lines are for standard algorithms and the full lines are for their secure-federated versions using SAMBA. The last plot shows the share in the execution time for each node in SAMBA, with $K = 10$ and budget $N = 10^3$.


Radu Ciucanu (PhD, Univ. Lille 1 & Inria, 2015) is an associate professor at INSA Centre Val de Loire & LIFo, working mainly on secure outsourced data analytics.

Pascal Lafourcade (PhD, ENS Cachan & LSV, 2006) is an associate professor at Univ. Clermont Auvergne & LIMOS, working mainly on cryptography and information systems security.

Gaël Marcadet (MSc, Univ. Orléans, 2021) is currently pursuing an internship on secure protocols for federated multi-armed bandits.

Marta Soare (PhD, Univ. Lille 1 & Inria, 2015) is an associate professor at Univ. Orléans & LIFO, working mainly on secure outsourced data analytics.
Appendix A
Security Details and Proofs

In Sect A.1, we detail the definition of the security tools introduced in section 3.2. We also introduce the proofs of security to ensure that SAMBA is secure, in section ??.

A.1 Additional Information on Security Tools

In this section, we detail the definition of the security tools briefly introduced in Sect. 3.2 in order to provide enough background to formally prove the security of our protocols. Before introducing the two cryptographic schemes, we point out that each of them has a security parameter $\lambda$ that is input to key generation. By $1^\lambda$ we denote the unary representation of $\lambda$, which is a standard notation in cryptography. Our security theorems are always asymptotic i.e., they describe the behavior when $\lambda$ becomes infinitely large. In practice, the security parameter is the length of the keys, for both Paillier and AES-GCM.

A.1.0.1 Paillier asymmetric encryption: Paillier’s cryptosystem [10] is an asymmetric partial homomorphic encryption scheme defined by a triple of polynomial-time algorithms $(G, E, D)$ and a security parameter $\lambda$ such that:

- $G(1^\lambda)$ generates two prime numbers $p$ and $q$ according to $\lambda$, sets $n = p \cdot q$ and $\Lambda = \text{lcm}(p - 1, q - 1)$ (i.e., the least common multiple), generates the group $(\mathbb{Z}_n^*, \cdot)$, randomly picks $g \in \mathbb{Z}_n^*$ such that $M = (L(g^\Lambda \mod n^2))^{-1} \mod n$ exists, with $L(x) = (x - 1)/n$. It sets $sk = (\Lambda, M)$, $pk = (n, g)$, it returns $(sk, pk)$.

- $E_{pk}(m)$ randomly picks $r \in \mathbb{Z}_n^*$, computes $c = g^m \cdot r^n \mod n^2$, and outputs $c$.

- $D_{sk}(c)$ computes $m = L(c^\Lambda \mod n^2) \cdot M \mod n$, and outputs $m$.

Paillier’s cryptosystem is additive homomorphic. Let $m_1$ and $m_2$ be two plaintexts in $\mathbb{Z}_n$. The product of the two associated ciphertexts with the public key $pk = (n, g)$, denoted $c_1 = E_{pk}(m_1) = g^{m_1} \cdot r_1^n \mod n^2$ and $c_2 = E_{pk}(m_2) = g^{m_2} \cdot r_2^n \mod n^2$, is the encryption of the sum of $m_1$ and $m_2$. Indeed, we have:

$$E_{pk}(m_1) \cdot E_{pk}(m_2) = c_1 \cdot c_2 \mod n^2$$

$$= (g^{m_1} \cdot r_1^n) \cdot (g^{m_2} \cdot r_2^n) \mod n^2$$

$$= (g^{m_1 + m_2} \cdot (r_1 \cdot r_2)^n) \mod n^2$$

$$= E_{pk}(m_1 + m_2).$$

A.1.0.2 AES-GCM symmetric encryption: AES [8] is a NIST standard for symmetric encryption that encrypts messages of 128 bits. To encrypt a message larger than 128 bits, AES is used as a block cipher and message’s blocks are mnage with mode, for instance with Galois Counter Mode (GCM) mode. Galois/Counter Mode (GCM) is a block cipher mode of operation, bases on CounTeR (CTR) mode, that uses universal hashing over a binary Galois field $\text{FG}(2^{128})$ to provide authenticated encryption. It can be implemented in hardware to achieve high speeds with low cost and low latency.

The AES-GCM cryptosystem belongs to the Authenticated Encryption with Associated Data (AEAD) family, which ensures confidentiality but also integrity and authenticity. It uses external authenticated data (e.g. packet header) to provide authenticated encryption and decryption operations. Is also an “online” cipher, in sense that is does not require AES-GCM to provide authenticated encryption. It can be implemented in hardware to achieve high speeds with low cost and low latency.

The AES-GCM cryptosystem is defined by a triple of polynomial-time algorithms (KeyGen, Enc, Dec) and a security parameter $\lambda$ such that:

- KeyGen($1^\lambda$) generates $K$, a uniformly random symmetric key of 128, 192 or 256 bits, according to $\lambda$.

- The Enc function takes 4 inputs: the secret key $K$ whose length is appropriate for the underlying block cipher, an initialization vector $IV$ that can have any number of bits between 1 and $2^{64}$ or 96 bits for performance concerns, the plaintext $P$ that has any number of bits between 0 and $2^{128} - 256$, and some Additional authenticated data (AAD) denoted by $A$. These data is authenticated, but not encrypted, and can have any number of bits between 0 and $2^{64}$. Enc function produces the ciphertext denoted $C$ of the same number of bits than $P$, and an authentication tag $T$ that can have a number of bits between 0 and 128.

- The Dec function takes 5 inputs: the same secret key $K$ whose length is appropriate for the underlying block cipher, the same initialization vector $IV$, the ciphertext $C$ produced by the Enc encryption function that has a number of bits equals to the plaintext $P$, the same Additional Authenticated Data (AAD) given to the Enc encryption function, and the authentication tag $T$ produced by the Enc encryption function. The Dec function produces as an output the plaintext $P$ or FAIL in case where the given ciphertext is not authentic.

Before to explain in details Enc and Dec functions, we introduce our notation based on Recommendation for Block Cipher Modes of Operation [9]. By $0^n$ (resp $1^n$), we define a $n$ bits string composed only by 0 (resp 1). The len($S$) function returns the length in bits of a $n$ bits string $S$ given in parameter. The $\text{MSB}_n(S)$ function returns only the $n$ most significant bits (leftmost) in the bits string $S$ given in parameter. By $\{\}$, we define the bit string with zero length. Moreover, we split $P$ into $n$ 128-bits strings $P_1, P_2, ..., P_{n-1}, P_n$ where $P_n$ has a number of bits $u$ between 0 and 128. As describe above, as the ciphertext $C$ has the same length of $P$, we split $C$ into $n$ 128-bits strings $C_1, C_2, ..., C_{n-1}, C_n$ where $C_n$ has a number of bits $u$ between 0 and 128. In the same way, we split Additional Authenticated Data (AAD) denoted $A$ into $m$ 128-bits strings $A_1, A_2, ..., A_{m-1}, A_m$ where $A_m$ has a number of bits $v$ between 0 and 128. Also, we denote by $t$ the number of
bits in the bits string $T$. Finally, for any $X, Y \in GF(2^{128})$, we denote by $X \cdot Y$ the multiplication of $X$ and $Y$ in the galois fields $FG(2^{128})$.

First, we detail the behavior of the Enc function.

$$H = E(K, 0^{128})$$

$$Y_0 = \begin{cases} IV \parallel 0^{31} \parallel 1 & \text{if } \text{len}(IV) = 96 \\ \text{GHASH}(H, \{\}, IV) & \text{otherwise} \end{cases}$$

$$Y_i = \text{incr}(Y_{i-1}) \text{for } i \in \{1, 2, \ldots, n\}$$

$$C_i = P_i \oplus E(K, Y_i) \text{for } i \in \{1, 2, \ldots, n-1\}$$

$$C_n^* = P_n^* \oplus \text{MSB}_4(E(K, Y_n))$$

$$T = \text{MSB}_4(\text{GHASH}(H, A, C) \oplus E(K, Y_0))$$

In details, $E$ is the block cipher which for a given secret key $K$ and a 128 bits plaintext block, produces the 128 bits ciphertext block. The counter values are generated using the function incr(), 32 bits of its argument as a nonnegative integer with the least significant bit on the right, and increments this value modulo $2^{32}$. More formally, the value of $\text{incr}(F \parallel I)$ is $F \parallel (I + 1 \mod 2^{32})$.

The function GHASH defined by $\text{GHASH}(H, A, C)$ returns the bits string $X_i$ with

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & \text{for } i \in \{1, \ldots, m-1\} \\ (X_{m-1} \oplus (A_m^* \mid 0^{128-v})) \cdot H & \text{for } i = m \\ (X_{i-1} \oplus C_i) \cdot H & \text{for } i \in \{m+1, \ldots, m+n-1\} \\ (X_{m+n-1} \oplus (C_m^* \mid 0^{128-u})) \cdot H & \text{for } i = m+n \\ (X_{m+n} \oplus (\text{len}(A) \mid \text{len}(C))) \cdot H & \text{for } i = m+n+1 \end{cases}$$

The authenticated decryption is defined as follows:

$$H = E(K, 0^{128})$$

$$Y_0 = \begin{cases} IV \parallel 0^{31} \parallel 1 & \text{if } \text{len}(IV) = 96 \\ \text{GHASH}(H, \{\}, IV) & \text{otherwise} \end{cases}$$

$$T' = \text{MSB}_4(\text{GHASH}(H, A, C) \oplus E(K, Y_0))$$

$$Y_i = \text{incr}(Y_{i-1}) \text{for } i \in \{1, 2, \ldots, n\}$$

$$P_i = C_i \oplus E(K, Y_i) \text{for } i \in \{1, 2, \ldots, n-1\}$$

$$P_n^* = C_n^* \oplus \text{MSB}_4(E(K, Y_n))$$

The decryption function returns $P$ only if the computed authentication tag $T'$ equals to $T$, else FAIL is returned.

A.1.0.3 IND-CPA (INDistinguishability under Chosen-Plaintext Attack) [29]: Let $\Pi = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt})$ be a cryptographic scheme. The probabilistic polynomial-time (PPT) adversary $A$ tries to break the security of $\Pi$. The IND-CPA game, denoted by $\text{EXP}(A)$, works as follows: the adversary $A$ chooses two messages $(m_0, m_1)$ and receives a challenge $c = \text{Encrypt}(LR_b(m_0, m_1))$ from the challenger who selects a bit $b \in \{0, 1\}$ uniformly at random, and where $LR_b(m_0, m_1)$ is equal to $m_0$ if $b=0$, and $m_1$ otherwise. The adversary, knowing $m_0, m_1$ and $c$, is allowed to perform any number of polynomial computations or encryptions of any messages, using the encryption oracle, in order to output a guess $b'$ of the encrypted message in $c$ chosen by the challenger. Intuitively, $\Pi$ is IND-CPA if there is no PPT adversary that can guess $b$ with a probability significantly better than $\frac{1}{2}$. By $\alpha = \Pr[b' \leftarrow \text{EXP}(A); b = b']$, we denote the probability that $A$ correctly outputs her guessed bit $b'$ when the bit chosen by the challenger in the experiment is $b$. A scheme is IND-CPA secure if $\alpha - \frac{1}{2}$ is negligible function in $\lambda$, where a function $\gamma$ is negligible in $\lambda$, denoted $\text{negl}(\lambda)$, if for every positive polynomial $p(\cdot)$ and sufficiently large $\lambda$, $\gamma(\lambda) < 1/p(\lambda)$.

Both cryptographic schemes mentioned earlier in this section are IND-CPA: (i) Paillier is IND-CPA under the decisional composite residuosity assumption [10], and (ii) AES-CCM is IND-CPA under the assumption that AES is a pseudo-random permutation [29].

In our theorems, the notion of “better than random” is consistent with the aforementioned IND-CPA property. We also point out an additional notation used in the proofs. Similarly to Landau Big O notation, where by convention $O(f)$ can describe any function bounded above by $f$, we abuse notation and denote by $\text{negl}(\lambda)$ any function negligible in $\lambda$. Notably, we have $\text{negl}(\lambda) + \text{negl}(\lambda) = \text{negl}(\lambda)$ and we may write $x + \text{negl}(\lambda)$ instead of $x - \text{negl}(\lambda)$.

A.1.0.4 IND-PERM (INDistinguishability under Permutation): Let a permutation $\sigma$ which, for a given list $L = [v_1, v_2, \ldots, v_n]$, returns the permuted list $\sigma(L) = \sigma([v_1, v_2, \ldots, v_n]) = [\sigma(v_1), \sigma(v_2), \ldots, \sigma(v_n)]$. The probabilistic polynomial-time (PPT) adversary $A$ tries to retrieve the initial list from the permuted one. The IND-PERM game, denoted by $\text{EXP}(A)$, works as follows: the adversary $A$ chooses $n$ values $v_1, v_2, \ldots, v_n$ sent to a challenger who build the list $L$ containing $v_1, v_2, \ldots, v_n$ in an arbitrary order. Then, it sends $\sigma(L)$ to the adversary, which by knowing $v_1, v_2, \ldots, v_n$ and $\sigma(L)$,
outputs a guess on $i_m$, the position of the higher value $v_{i_m}, i_m \in [n]$, in the initial list $L$. Intuitively, a permutation is IND-PERM if there is no PPT adversary able to guess $i_m$ with a probability significantly better than $\frac{1}{n}$.

By $\alpha = \Pr[i'_m \leftarrow \text{EXP}(A); i'_m = i_m]$, we denote the probability that $A$ correctly outputs her guessed index $i'_m$, when the real higher value index chosen in the initial list $L$ built by the challenger in the experiment is $i_m$. Notice that a pseudorandom permutation is IND-PERM, as there is a probability $\frac{1}{n}$ to guess correctly the index $i_m$. We show an example of a game construction done by the Challenger with $n = 2$ in Fig. 8.

### A.2 Security Proofs for Sect. 5.4

In this section, we provide formal statements and proofs for the security properties of SAMBA that we have already outlined in Sect. 5.4. In Table 9 we summarize what each participant in SAMBA knows/does not know, with pointers to the relevant theorems.

<table>
<thead>
<tr>
<th>Data</th>
<th>Participant</th>
<th>DO$_i$</th>
<th>DC</th>
<th>Comp</th>
<th>Controller</th>
<th>Ext</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cumulative reward</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum of rewards for DO$_i$</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum of rewards for DO$_j \neq i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arm pulled at time step $t$</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Reward at time step $t$</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Fig. 9. The ✓ should be read as: the participant can see in clear the concerned piece of data, whereas an empty case means the opposite. The * should be read as: only if DO$_i$ is pulled at time step $t$. Ext should be read as: an external network observer having access to all messages exchanged between participants.

Before formally stating the theorems, we recall (cf. Sect. 5.4) that the participants are honest-but-curious and do not collude. We also introduce some notations needed for the theorem statements:

- $n_{i,t}$ = the number of times arm $i$ has been pulled until round $t$.
- $s_{i,t}$ = the sum of rewards obtained by arm $i$ until round $t$.
- $\text{data}_A$ = the data to which participant $A$ has access until round $t$, where $A$ can be a participant from Fig. 5 or the external observer (ext). If $t$ is omitted, this denotes the data to which $A$ has access at the end of the protocol.
- $A^{pb()}(d)$ = the answer of a Probabilistic Polynomial-Time (PPT) adversary $A$ that knows $d$ and tries to solve the problem $pb$. Depending on the problem, $pb$ can also take some input.
- By negligible in $\lambda$, we denote that our security theorems are always asymptotic i.e., they describe the behavior when the security parameter $\lambda$ of the cryptographic schemes becomes infinitely large.

We next provide theorems that state each non-trivial property from Table 9.

**Lemma 1.** Let $A$ be a PPT adversary trying to find the cumulative reward $R$, and let $B$ be a PPT adversary trying to find the sum of rewards of some arm. Let $d$ be some data, $\text{cr}(.)$ be the problem of guessing the cumulative reward, and $\text{sum}(.)$ be the problem of guessing the sum of rewards of some arm. We have the following statement: $A^{\text{cr}(.)}(d)$ has a non-negligible advantage $\Rightarrow B^{\text{sum}(.)}(d)$ has a non-negligible advantage.

**Proof.** Assume that $B$ can guess the sum of rewards of some arm with probability better than random. Then, $A$ can call $B$, and hence get the sum of rewards of one arm with probability better than random. From this sum, $A$ can guess
Theorem 2. For an arm $s$ during steps $\lambda$ in architecture involves communication, we must prove the security for each entity. AMBA In this section, we prove the security of each entity implied in S\textsuperscript{A}.2.1 Security in Informed architecture context

of rewards of some arm. IND-CPA property of Paillier’s cryptosystem [10]. Assume there exists a PPT adversary $E$ values $B$ how to construct an adversary $E$ wins the IND-CPA game, in which $E$ answers randomly and is correct with probability $1/2$. Otherwise (probability $1/2$), let us consider the value of $b$. 

- If $b = 0$ (probability $1/2$), then we have two cases:
  * If the output of $A$ is $(1, m_1)$ (probability $p_S(s_{1,t})$), then $B$ answers 1 and it is wrong, hence the probability of success is 0.
  * Otherwise (probability $1 - p_S(s_{1,t})$), $B$ answers randomly and is correct with probability $1/2$. The probability of success of this branch is $\frac{1}{2}(1 - p_S(s_{1,t}))$. 

- If $b = 1$ (probability $1/2$), then we have two cases:
  * If the output of $A$ is $(1, m_1)$ (probability $p_S(s_{1,t}) + x + \text{negl(}\lambda\text{)})$, then $B$ correctly answers 1. The probability of success of this branch is $\frac{1}{2}(p_S(s_{1,t}) + x + \text{negl(}\lambda\text{)))$.
  * Otherwise (probability $1 - p_S(s_{1,t}) - x - \text{negl(}\lambda\text{)})$, $B$ answers randomly and is correct with probability $1/2$. The probability of success of this branch is $\frac{1}{2}(1 - p_S(s_{1,t}) - x - \text{negl(}\lambda\text{)))$. 

Proof. During steps $2 - 5$, all transmitted data over the Controller are encrypted by using a symmetric key only known by DO\textsubscript{1} and Comp. Hence, Controller does not see any data in clear. At the end of S\textsubscript{AMBA}, at Step 6, Controller receives the values $\mathcal{E}_\text{DC}(s_{1,t}), \ldots, \mathcal{E}_\text{DC}(s_{K,t})$. We prove that retrieving any information about any $s_{i,t}$ from these ciphertexts breaks the IND-CPA property of Paillier’s cryptosystem [10]. Assume there exists a PPT adversary $A$ able, from data\textsubscript{Controller} to find $s_{i,t}$ for some $i$ with non negligible advantage $x$:

$$\text{Pr}(i, s_{i,t} \leftarrow A_{\text{sum}(t)}(\text{data}_\text{Controller}^t); \hat{s}_{i,t} = s_{i,t} - p_S(s_{i,t}))$$

is negligible in $\lambda$, where $A_{\text{sum}(t)}(\text{data}_\text{Controller}^t)$ returns $(i, \hat{s}_{i,t})$ in which $\hat{s}_{i,t}$ is $A$’s guess on $s_{i,t}$ for the arm $i$ (chosen by $A$), and $p_S(s_{i,t})$ is the probability of obtaining a sum of rewards $s_{i,t}$ for an arm $i$ at time step $t$.

In the worst case, each $i \in [K]$ has an equal probability of being chosen by $A$. We also assume that if data\textsubscript{Controller} does not correspond to the data collected by Controller during a run of S\textsubscript{AMBA} (for instance, if one piece of data\textsubscript{Controller} has been replaced by another unrelated message), then $A$ does not give any advantage. If such an adversary $A$ exists, then we show how to construct an adversary $B$ able to break the IND-CPA property of Paillier.

Let us build an IND-CPA game, in which $B$ chooses two values $m_0, m_1$, and sends them to the challenger. The challenger randomly selects $b \in \{0, 1\}$ and answers with $\mathcal{E}_\text{DC}(m_b)$. $B$ wins the IND-CPA game if $B$ guesses $b$ with a non-negligible advantage.

To do so, $B$ first creates a simulation of a S\textsubscript{AMBA} execution i.e., $B$ creates nodes DC’, Controller’, DO’\textsubscript{t}, with Bernoulli distributions defined by $\mu’$, of its choice. Then, $B$ runs an execution of S\textsubscript{AMBA} on these nodes. Because $B$ controls all the nodes, it knows the sums of rewards $s’_{1,t}, \ldots, s’_{K,t}$.

As input for the IND-CPA game, $B$ chooses $m_1 = s’_{1,t}$ and another value $m_0$, different from all $s’_{i,t}$, sends both values to the challenger, and receives $\mathcal{E}_\text{DC}(m_b)$. Then, $B$ computes $\mathcal{E}_\text{DC}(s’_{i,t})$ for each $i$, and calls $A_{\text{sum}(t)}(\mathcal{E}_\text{DC}(m_b), \mathcal{E}_\text{DC}(s’_{2,t}), \ldots, \mathcal{E}_\text{DC}(s’_{K,t}), L)$. 

The strategy of $B$ is as follows: if $A$ returns $(1, m_1)$, then $B$ answers 1. Otherwise, $B$ answers randomly. We next derive the probability of a correct answer by $B$.

- If $i \neq 1$ (probability $1 - \frac{1}{K}$), then $B$ answers randomly and is correct with probability $1/2$. Hence this branch offers a probability of success of $(1 - \frac{1}{K})\frac{1}{2}$.
- If $i = 1$ (probability $\frac{1}{K}$), let us consider the value of $b$.
  - If $b = 0$ (probability $\frac{1}{2}$), then we have two cases:
    * If the output of $A$ is $(1, m_1)$ (probability $p_S(s_{1,t}))$, then $B$ answers 1 and it is wrong, hence the probability of success is 0.
    * Otherwise (probability $1 - p_S(s_{1,t}))$, $B$ answers randomly and is correct with probability $1/2$. The probability of success of this branch is $\frac{1}{2}(1 - p_S(s_{1,t}))$.
  - If $b = 1$ (probability $\frac{1}{2}$), then we have two cases:
    * If the output of $A$ is $(1, m_1)$ (probability $p_S(s_{1,t}) + x + \text{negl(}\lambda\text{))$, then $B$ correctly answers 1. The probability of success of this branch is $\frac{1}{2}(p_S(s_{1,t}) + x + \text{negl(}\lambda\text{)))$.
    * Otherwise (probability $1 - p_S(s_{1,t}) - x - \text{negl(}\lambda\text{)})$, $B$ answers randomly and is correct with probability $1/2$. The probability of success of this branch is $\frac{1}{2}(1 - p_S(s_{1,t}) - x - \text{negl(}\lambda\text{)))$. 

a lower bound on the cumulative reward, hence eliminating some possibilities, and thus guessing the cumulative reward with probability better than random.

⇒ If $A$ can guess the cumulative reward with probability better than random, then $B$ can use this cumulative reward as an upper bound on the sum of rewards of some arm, thus having a probability better than random of guessing the sum of rewards of some arm.

A.2.1 Security in Informed architecture context

In this section, we prove the security of each entity implied in S\textsubscript{AMBA} in the Informed architecture context. As this architecture involves communication, we must prove the security for each entity.
By aggregating the aforementioned cases, the probability $\alpha$ of success of $B$ is:

$$\alpha = \left(1 - \frac{1}{K} \right) \frac{1}{2} + \frac{1}{K} \left(1 - ps(s_{1,t})\right) \frac{1}{2} + \frac{1}{2} \left(1 - \frac{1}{K} \right) \left(1 - ps(s_{1,t}) - x + \negl(\lambda)\right)$$

$$+ \frac{1}{2} \left(1 - \frac{1}{K} \right) \left(1 - ps(s_{1,t}) - x - \negl(\lambda)\right) \frac{1}{2}$$

$$= \frac{1}{2} - \frac{1}{2K} + \frac{1}{4K} - \frac{ps(s_{1,t})}{4K} + \frac{ps(s_{1,t})}{2K} + \frac{x}{2K} + \frac{1}{4K} - \frac{ps(s_{1,t})}{4K} - \frac{x}{4K} + \negl(\lambda)$$

$$= \frac{1}{2} + \frac{x}{4K} + \negl(\lambda)$$

Hence, $B$ has an advantage of $\frac{1}{2K}$ in the IND-CPA game, which is non negligible. This is a contradiction with the fact that Paillier is IND-CPA secure. Consequently, there does not exist any PPT adversary $A$ that violates the property stated in the theorem.

As a corollary, by Lemma 1 and Theorem 2, we infer that Controller cannot learn the cumulative reward with probability better than random.

A.2.1.2 Security of DO: By construction of SAMBA, each arm node DO$_i$ knows its sum of rewards $s_i$. In addition, at turn when it is not drawn, it cannot predict the drawn arm DO$_{i_m}$ (Theorem 3). Finally, it cannot learn the sum of rewards of any other arm (Theorem 4).

**Theorem 3.** At the end of a time step $t \in [K + 1, N]$, given data$_{DO_i}$ and a drawn arm DO$_{i_m}$ where $i \neq i_m$, the DO$_i$ cannot predict which arm has been drawn with a probability better than random.

**Proof.** Let suppose at time step $t \in [K + 1, N]$ a drawn arm DO$_{i_m}$ and another DO$_i$ with $i \neq i_m$. After step 5, the data$_{DO_i}$ contains scores $v_{i,1}, \ldots, v_{i,t}$ and the list of received selection bit $b_{i,1}, \ldots, b_{i,t}$. Hence, it does not have any information about the other DO$_j$. Thus at a given $t \in [N]$, DO$_i$ cannot predict the pulled arm DO$_{i_m}$ with a probability better than random, i.e. $\frac{1}{K-1}$.

**Theorem 4.** For an arm $i \in [K]$ and a time step $t \in [K + 1, N]$, an honest-but-curious DO$_i$ cannot learn $s_{j,t}$ for some other arm $j \neq i$, given data$_{DO_i}$, with a probability better than random. More precisely, for all PPT adversaries $A$,

$$\Pr \left( (j, s_{j,t}) \leftarrow A^{sum(t)}(data_{DO_i}^t); \hat{s}_{j,t} = s_{j,t} \right) - p_R(n_{i,t}, t, s_{j,t}) \right)$$

is negligible in $\lambda$, where $A^{sum(t)}(data_{DO_i}^t)$ returns a tuple $(j, s_{j,t})$ in which $j \neq i$ is chosen by $A$ and $\hat{s}_{j,t}$ is $A$’s guess of the sum of rewards for arm $j$, and $p_R(n_{i,t}, t, s_{j,t})$ is the probability of arm $j$ to have sum of rewards $s_{j,t}$ at round $t$ seen that arm $i$ has been pulled $n_{i,t}$ times.

**Proof.** During steps 2-5, a DO$_i$ does not leak any data and does not receive information about the other arm. In a such way, at turn $t \in [N]$, the DO$_i$ does not have non negligible advantage to retrieve the partial cumulative reward $s_{j,t}$ with $j \neq i$. Thus, for any DO$_i$ and a DO$_j$ with $j \neq i$, DO$_i$ cannot predict $s_{j,t}$ at time step $t \in [N]$ better than random.

As a corollary, by Lemma 1 and Theorem 4 we infer that DO$_i$ cannot learn the cumulative reward $R$ with probability better than random.

A.2.1.3 Security of Comp: The Comp node is only involved at steps 3 and 4 to receive the permuted scores produced by each DO$_i$, select an arm based on these scores and returns the permuted list of selection bits. We prove that at a given $t \in [N]$, Comp cannot predict the cumulative reward of an arm $i \in [N]$.

**Theorem 5.** At the end of a time step $t \in [K + 1, N]$, given data$_{Comp}$, the Comp cannot predict which arm has been drawn with a probability better than random. More precisely, for all PPT adversaries $A$,

$$\Pr \left( \hat{i}_m \leftarrow A^{sum(t)}(data_{Comp}^t); \hat{i}_m = i_m \right) - \frac{1}{K}$$

is negligible in $K$ the number of arms, where $A^{sum(t)}(data_{Comp}^t)$ returns a guess $\hat{i}_m$ on the index $i_m$ of the pulled arm, and $\frac{1}{K}$ is the probability to guess the pulled arm at time step $t$ randomly.

**Proof.** At step 3, Comp receives a permuted list of encrypted masked values, able to decrypt them by knowing the key. Hence Comp is able to see in clear (but masked) values transmitted by each DO$_i$. The permutation $\sigma_t$, computed by Controller, does not allow Comp to define which arm is the next pulled arm. Otherwise, Comp would be able to define which arm has been pulled at each iteration of the protocol with a probability better than random. We prove that retrieving the pulled arm from the permuted list $\sigma_t(\{ov_{i,t}\}_{i \in [K]})$ breaks the IND-PERM property of the permutation $\sigma_t$. Assume there exists a PPT adversary $A$ able, from data$_{Comp}^t$ to find the pulled arm $i_m$ for some $i$ with non negligible advantage $x$:

$$\Pr \left( \hat{i}_m \leftarrow A^{sum(t)}(data_{Comp}^t); \hat{i}_m = i_m \right) - \frac{1}{K} = x + \negl(K).$$
In the worst case, each $i \in [K]$ has an equal probability of being chosen by $A$. We also assume that if $data^i_{\text{Comp}}$ does not correspond to the data collected by Comp during a run of SAMBA (for instance, if one piece of $data^i_{\text{Comp}}$ has been replaced by another unrelated message), then $A$ does not give any advantage. If such an adversary $A$ exists, then we show how to construct an adversary $B$ able to break the IND-CPA property of the permutation.

Let us build an IND-CPA game, in which $B$ chooses $n$ values $v_1, v_2, \ldots, v_n$, and sends them to the challenger. The challenger computes $L$ the initial list containing $v_1, v_2, \ldots, v_n$ in an arbitrary order. Then, it creates a permutation $\sigma$, and sends the permuted initial list $\sigma(L)$ to $B$. $B$ wins the IND-CPA game if $B$ guesses the index $i_m$ of the higher value $v_{i_m}$ in the initial list $L$, with a non-negligible advantage.

To do so, $B$ first creates a simulation of a SAMBA execution i.e., $B$ creates nodes $DC^t$, Controller', DO'$_t$, with Bernoulli distributions defined by $\mu^t$ of its choice. Then, $B$ runs an execution of SAMBA on these nodes. Because $B$ controls all the nodes, it knows the scores $v_1, v_2, \ldots, v_K$ produced by the $K$ data owners.

As input for the IND-CPA game, $B$ sends the scores $v_1, v_2, \ldots, v_K$ to the challenger, and receives a permuted list of scores $\sigma(L)$. Then, $B$ calls $A_{\text{sum}}(\sigma(L))$.

The strategy of $B$ is as follows: $B$ a guess on the best value index $i_m$ that $B$ will send to the Challenger, enjoying the non-negligible advantage $x$ of adversary $A$.

Hence, thanks to the adversary $A$, $B$ gets a non-negligible advantage $x + \text{negl}(K)$ to guess the index $i_m$ of the best value in $L$, thus $B$ can win to the IND-CPA game. This contradicts the fact that we use a pseudo-random permutation which is IND-CPA. Hence, we conclude that there does not exist any PPT adversary $A$ that violates the property stated in the theorem.

**Theorem 6.** At a time step $t \in [K + 1, N]$, an honest-but-curious Comp cannot learn $s_{i,t}$ of an arm $i \in [K]$, with probability better than random. More precisely, for all PPT adversaries $A$,

$$\left| \Pr \left[ (i,t,s_{i,t}) \leftarrow A_{\text{sum}}(t)(data^i_{\text{Comp}}); \hat{s}_{i,t} = s_{i,t} \right] - p_C(t,s_{i,t}) \right|$$

is negligible in $\lambda$, where $A_{\text{sum}}(t)(data^i_{\text{Comp}})$ returns a tuple $(j, \hat{s}_{j,t})$ in which $j \neq i$ is chosen by $A$ and $\hat{s}_{j,t}$ is $A$’s guess of the sum of rewards for arm $i$, and $p_R(t,s_{i,t})$ is the probability of arm $i$ to have sum of rewards $s_{i,t}$ at round $t$.

**Proof.** At step 2, each DO$_t$ computes two values: its score $v_t$ returned by the ComputeScore abstract function, and a mask $\alpha_t$. Tank to the seed$_\alpha$, all DO$_t$ have the same mask at a given $t \in [N]$. At the end of the step, each DO$_t$ sends the encrypted $\alpha_t v_t$ to the Controller, which builds and permutes the $K$ sized list with these values. The Comp node is able to see each $\alpha_t v_t$ in clear but does not have neither seed$_\alpha$ and $\alpha_t$. Hence, it cannot retrieve any of these data values except a permuted order of these values. Without priors, Comp cannot predict the partial cumulative reward $s_i$ for an arm $i \in [K]$ better than random.

A.2.1.4 Security of an external observer: An external observer sees all messages exchanged between nodes, from which we show that she cannot learn which arm is pulled at some round (Theorem 7) or the sum of rewards for some arm (Theorem 8).

**Theorem 7.** At each time step $t \in [K + 1, N]$, an honest-but-curious external observer cannot learn which arm is pulled at round $t$, given $data^t_{\text{ext}}$, with probability better than random. More precisely, for all PPT adversaries $A$,

$$\left| \Pr[Am(t)(data^t_{\text{ext}}) = i^t_{m}] - \frac{1}{K} \right|$$

is negligible in $\lambda$, where $Am(t)(data^t_{\text{ext}})$ returns the guess of $A$ on which arm is pulled at round $t$, and $i^t_{m}$ is the true arm pulled at round $t$.

**Proof.** By construction of SAMBA, all arms are pulled at the end of step 1. At each time step one arm is pulled, hence as $K$ arms are pulled at the beginning, the time step $t$ is set at $K + 1$ after the initial pulling. When the core of the protocol begins, at time step $t = K + 1$ and until the end of SAMBA i.e., time step $N$, there is a single arm pulled at each round. We next show that if there exists a PPT adversary with a non-negligible advantage in guessing the arm pulled at some time step $K + 1 \leq t \leq N$, then this would break the IND-CPA property of AES-GCM.

An external observer (denoted $ext$ in the sequel) sees all encrypted messages that are exchanged among SAMBA participants. We denote by $data^t_{\text{ext}}$ this collection of data after round $t$. We assume, toward a contradiction, that there exists a PPT adversary $A$ able from $data_{\text{ext}}$ to find the arm $i^t_{m}$ pulled at some round $t$ with a non-negligible advantage $x$,

$$\left| \Pr[Am(t)(data^t_{\text{ext}}) = i^t_{m}] - \frac{1}{K} \right| = x + \text{negl}(\lambda).$$

We also assume that if $data^t_{\text{ext}}$ does not correspond to an actual collection of encrypted messages that $ext$ sees, then the advantage for such an input is negligible.

We next show that by using the adversary $A$, we can construct an adversary $B$ able to break the IND-CPA property of AES-GCM. To do so, $B$ creates a simulation of an SAMBA execution, similarly to the proof of Theorem 7. Even though the messages of such a simulation are encrypted, $B$ knows the keys hence the state of each arm. In particular, $B$ knows in plain text the messages sent at step 5 by the Controller to each DO$_t$ at a given time step $t$. To recall, this message contains the
collection of encrypted messages from the B

if Enc

guess a bit about m

minimal when

Theorem 8.

is IND-CPA secure. Hence, we conclude that there does not exist any PPT adversary B

Hence, A returns the correct im, then B returns 1, otherwise answer randomly.

- If b = 0 (probability 1/2), then A does not receive a correct simulation because no arm is pulled at round t. According to our assumption, A does not give any advantage.
  - If A returns the correct im (probability 1/K), then B answers 1 and is wrong.
  - Otherwise (probability 1 − 1/K), then B answers randomly and is correct with probability 1/2. This branch yields a probability of success of 1/2(1 − 1/K)^2.

- If b = 1 (probability 1/2), then the advantage given by A can be leveraged by B.
  - If A returns the correct i_m (probability 1/K + x + negl(λ)), then B correctly answers 1. The probability of success of this branch is 1(1/K + x + negl(λ)).
  - Otherwise (probability 1 − 1/K − x − negl(λ)), B answers randomly and is correct with probability 1/2. This branch yields a probability of success of 1/2(1 − 1/K − x − negl(λ))^2.

By aggregating the aforementioned cases, the probability α of success of B is:

\[ \alpha = \frac{1}{2} \left( 1 - \frac{1}{K} \right)^2 + \frac{1}{2} \left( 1 + x + \text{negl}(\lambda) \right) + \frac{1}{2} \left( 1 - 1/K - x - \text{negl}(\lambda) \right)^2 \]

\[ = \frac{1}{4} - \frac{1}{2K} + \frac{1}{2} + \frac{1}{2} - \frac{1}{4K} - \frac{x}{4} + \text{negl}(\lambda) \]

\[ = \frac{1}{2} + \frac{x}{4} + \text{negl}(\lambda) \]

Hence, B has an advantage of \( \frac{1}{4} \) in the IND-CPA game, which is non negligible. This contradicts the fact that AES-GCM is IND-CPA secure. Hence, we conclude that there does not exist any PPT adversary A that violates the property stated in the theorem.

\[ \square \]

Theorem 8. For an arm i \( \in \llbracket K \rrbracket \) and a time step t \( \in \llbracket N \rrbracket \), an honest-but-curious external observer cannot learn s_{i,t}, given data_{ext}', with a probability better than random. More precisely, for all PPT adversaries A,

\[ \Pr \left[ (i, \hat{s}_{i,t}) \leftarrow A^{\text{arm}(t)}(\text{data}_{\text{ext}}'); \hat{s}_{i,t} = s_{i,t} \right] - p_Q(t, s_{i,t}) \]

is negligible in λ, where A^{arm(t)}(data_{ext}') returns (i, \hat{s}_{i,t}) in which \( \hat{s}_{i,t} \) is A's guess on s_{i,t} for the arm i (chosen by A), and p_Q(t, s_{i,t}) is the probability of obtaining a sum of rewards s_{i,t} from at most t pulls of arm i until round t.

Proof. The external observer collects data_{ext}', which consists of several encrypted messages, some of them being encrypted with Enc (AES-GCM) and some other being encrypted with Enc_P (Paillier). We prove that these messages do not provide an advantage bigger than an advantage of an adversary in a classical IND-CPA game on Enc or Enc_P. For simplicity, we assume that the data_{ext}' only contains two encrypted messages, Enc(m) and Enc_P(m). The proof can obviously be adapted if data_{ext}' consists of more than two messages.

The goal of the adversary is to extract at least a bit of information from either m or n. The entropy of this system is minimal when m = n. Hence, when m = n, the adversary has the highest probability of guessing at least a bit from at least m or n (which are the same in this case). As a consequence, in the general case, the advantage of an adversary having to guess a bit about m or n, knowing Enc(m) or Enc_P(m) is bounded above by the advantage of an adversary having to guess a bit about m, knowing Enc(m) and Enc_P(m).

Let us prove that the advantage of a PPT adversary in this latter case (having to guess a bit about m from Enc(m) and Enc_P(m)) is negligible. We assume toward a contradiction, that there exists a PPT adversary A able to win the game where, given Enc(m) and Enc_P(m), A recovers a bit of information about m with a non-negligible advantage x: given Enc(m) and Enc_P(m), the probability that A outputs a correct guess about a bit of m is equal to 1/2 + x + negl(λ).

We use this adversary to create another adversary B able to break the IND-CPA property of the encryption schemes Enc (or Enc_P, respectively). As usually in the IND-CPA game, B chooses two messages m_0 and m_1, and sends them to the challenger. Then, B receives the challenge Enc(m_0) (or Enc_P(m_0), respectively), and calls A(Enc(m_0), Enc_P(m_0)) (or A(Enc_P(m_0), Enc(m_0)), respectively). If A returns a correct guess about m_0, then B returns 0. Otherwise, it returns 1.

- If b = 0 (happens with probability 1/2), then A has a non negligible advantage in guessing a bit about m.
  - A outputs a correct guess about a bit of m_0 with probability 1/2 + x + negl(λ). In this case, B is correct. This branch happens with probability 1/2(1/2 + x + negl(λ)).
  - If A does not answer correctly (happens with probability 1/2 − x − negl(λ)), then A is correct with probability 1/2. This branch happens with probability 1/2(1/2 − x − negl(λ))^2.

- If b = 1 (happens with probability 1/2), then A has no advantage.
  - If A returns a correct guess about one bit of m_0 (happens with probability 1/2), then B is wrong.
arm probability distributions are equiprobable to DC. Similarly, we observe that the data client does not receive any message until the end of SAMBA (Step 7). By construction of SAMBA, all arms are initially pulled once, then from time step $K$ and until the end of SAMBA i.e., time step $N$, there is a single arm pulled at each time step. In particular, the data client does not receive any information on which arm is pulled at some round, hence her best strategy is to answer randomly, with a probability of success of $\frac{1}{2}$. Hence, $B$ has a non-negligible advantage of $\frac{1}{4}x$ in the IND-CPA game against Enc (or $E_{DC}$, respectively), which is a contradiction with its IND-CPA property. Guessing a bit about the encrypted message is equivalent to guessing the reward with a probability better than random (i.e., better than $p_{Q}(t,s_{i,t})$ cf. our theorem statement), which concludes our proof.

As a corollary, by Lemma 1 and Theorem 8, we infer that the external observer cannot learn the cumulative reward with probability better than random.

A.2.1.5 Security of DC: The data client knows the cumulative reward that she can decrypt after Step 7. Moreover, the data client cannot learn the arms selected at some round (Theorem 9) or the sum of rewards for some arm (Theorem 10).

**Theorem 9.** For each time step $t \in [K+1,N]$, the data client DC cannot guess which arm is pulled at time step $t$ with probability better than random.

**Proof.** The data client does not receive any message until the end of SAMBA (Step 7). By construction of SAMBA, all arms are initially pulled once, then from time step $K$ and until the end of SAMBA i.e., time step $N$, there is a single arm pulled at each time step. In particular, the data client does not receive any information on which arm is pulled at some round, hence her best strategy is to answer randomly, with a probability of success of $\frac{1}{2}$.

**Theorem 10.** For an arm $i \in [K]$, the data client DC cannot guess the sum $s_{i}$ of rewards for the arm $i$ with probability better than random.

**Proof.** Similarly to the previous proof, we observe that the data client DC does not receive any message until the end of SAMBA (Step 7). In particular, DC does not get any information about which arm is selected at some round. Because all arm probability distributions are equiprobable to DC, it is also true that all partitions of the cumulative reward $R$ are equiprobable to DC, thus DC has no advantage in guessing the partition of rewards. Hence, the probability of DC guessing a correct partition of the rewards is equal to $\frac{1}{P(R)}$, where $P(R)$ is the number of partitions of $R$. This observation also proves that DC cannot guess the individual sum of rewards of some arm $i$. If it was the case, then DC would know that some of the partitions are more likely e.g., if DC can guess the sum of rewards $s_{i}$ of the arm $i$, then all partitions not having $s_{i}$ as the value for arm $i$ would be discarded, which is a contradiction.

**A.2.1.6 Impact of collusions:** As pointed out earlier, an hypothesis behind our security theorems is that cloud nodes do not collude. By collusion we mean that cloud nodes put together all their data. If at least 2 of the DO nodes collude, they could learn their respective algorithm inputs (i.e., bandit arm values that only the data owner is supposed to know at the same time) and outputs (i.e., cloud nodes could sum up the partial sums of rewards known by each node), hence SAMBA would not satisfy the desirable security properties. In addition to following a standard security model (cf. discussion in Sect. 2.1), we believe that the no-collusion hypothesis is necessary if we want a secure cumulative reward maximization algorithm that produces exactly the same output as the chosen standard algorithm version. Indeed, as already mentioned in the introduction, it is not currently possible in practice to use fully-homomorphic encryption on real numbers without result approximation. Hence, to minimize data leakage, our choice is to do computations on real numbers in clear, and to distribute reward functions and scores computations among $K$ cloud nodes (one per arm), each of them having access in clear only to data pertaining to its arm.

**A.2.1.7 Number of computations rounds:** Our protocol allows for a given algorithm to perform one or more computation rounds. We prove earlier that both Random and Informed architectures are secure and do not allow leaking data in a single computation, but does it the same over several ones? By construction, our weak point is Comp which can see in clear the masked values produced by each DO. Since the mask is updated at each iteration, then suppose that at a given time step $t$ composed by two computation rounds, and that the mask leaks due to predictable value computed by a DO. Hence, the privacy of the generated value is already broken. At the next computation round, since the mask is updated, the masked value cannot be retrieved.

**APPENDIX B**

**IMPLEMENTATION**

We propose in this section the implementation of SAMBA version of algorithms presented in this paper. Our protocol uses several seed to work. We recall that a seed allows to generate a random series of deterministic values $v_{1}, v_{2}, \ldots, v_{t}$, and
def SelectArchitecture(t, l, A)
  (ε, seedε) ← Parameters from A
  if nextReal(0, 1, seedε, t) ≤ ε
    return Random
  else
    return Informed

def ComputeScore(t, l, s, n, A)
  return \frac{s}{n}

def SelectArm(t, l, σ_{i}(α, \frac{n}{n_{i}}) \in \mathcal{K}), A)
  return arg max \sigma_{i}(α, \frac{n}{n_{i}}) \in \mathcal{K}

def HandleSelect(t, l, b, s, n, A)
  if b = 1
    s_i ← s_i + pull(i)
    n_i ← n_i + 1
  return (s_i, n_i)

(a) The ε-greedy implementation

(b) The ε-decreasing-greedy implementation

Fig. 10. ε-greedy and ε-decreasing-greedy

subsequently to compute the same random value at a given time step t, useful to generate the same value in several independent servers without extra communications.

- pull(i) is the function which pulls the arm i and returns either 0 (no reward) or 1 (a reward).
- nextInt(a, b, seed, t) is the function which for a given seed, returns the t-th integer i ∈ [a, b].
- nextReal(a, b, seed, t) is the function which for a given seed, returns the t-th real i ∈ [a, b].
- probMatchingInt(a, b, weights) is the function which returns an integer i ∈ [a, b] where each i has a probability \in [0, 1] to be returned. Notice that |[a, b]| = \|weights\|.

B.0.1 ε-greedy (Fig. 10(a)) and ε-decreasing-greedy (Fig. 10(b))
The presented implementation is used to simulate ε-greedy algorithm. Notice the two extra seeds seedε used respectively in SelectArchitecture and SelectArm functions. As exposed in Section ??, we recall that these two functions must be deterministic. It implies that for a given time step and computation round, the output must be always the same. However, ε-greedy selects his behavior randomly with a probability ε. Hence, we use two seed which allows to render these functions deterministic.

As equals as the ε-greedy algorithm, ε-decreasing-greedy uses two seeds seedε and seedarm. These algorithms differs only in the ε definition, which is a constant in ε-greedy and decreasing in ε-decreasing-greedy.

APPENDIX C
DETAILS ABOUT SAMBA
C.1 Formal Mask Definition
In the protocol, we defined a mask used in Informed architecture to ensure privacy on scores computed by each DO. We formally define a mask by \alpha ∈ \mathbb{R}^*, respecting the so-called order-preserving and proportion-preserving properties, defines as follows:

Lemma 2. Let a mask \alpha ∈ \mathbb{R}^* and v_1, v_2 ∈ \mathbb{R} with v_1 ≤ v_2. We state that \alpha is order-preserving if we have \alpha v_1 ≤ \alpha v_2.
Proof. Let a mask \alpha ∈ \mathbb{R}^* and v_1, v_2 ∈ \mathbb{R}.
Case v_1 < v_2: We multiply v_1 and v_2 by \alpha to obtain \alpha v_1 < \alpha v_2.
Case v_1 > v_2: We multiply v_1 and v_2 by \alpha to obtain \alpha v_1 > \alpha v_2.
Case v_1 = v_2: We multiply v_1 and v_2 by \alpha to obtain \alpha v_1 = \alpha v_2.

Lemma 3. Let a mask \alpha ∈ \mathbb{R}^* and v_1, v_2 ∈ \mathbb{R}. We state that \alpha is proportion-preserving if we have \frac{v_1}{\alpha v_1 + \alpha v_2} = \frac{v_1}{v_1 + v_2} and \frac{v_2}{\alpha v_1 + \alpha v_2} = \frac{v_2}{v_1 + v_2}.
Proof. Let a mask \alpha ∈ \mathbb{R}^* and v_1, v_2 ∈ \mathbb{R}. Let the initial proportions \frac{p_1}{\alpha v_1 + \alpha v_2}, \frac{p_2}{v_1 + v_2} and let \frac{p'_1}{\alpha v_1 + \alpha v_2}, \frac{p'_2}{v_1 + v_2} the initial proportions where v_1, v_2 have been masked by \alpha.
Proof. To prove the theorem, we do a proof by recurrence over the number of computation rounds $nbIteration_{\mathcal{A}}$.

C.2 Formal statement of abstract functions

We regroup in this table each abstract function introduced in the protocol in Fig.14.

C.3 Detailed Explanation about the Reinforcement Comparison Algorithm

It is similar to Softmax in that it selects the next arm drawn with a probability matching, where each arm has a probability $p_{i,t}$ of being drawn. In details, it maintains a preference $\pi_{i,t}$ for each arm $i$, which requires two learning rates $\alpha, \beta \in [0, 1]$. At each time step $t$, the probability to draw an arm $i$ is computed as $p_{i,t} = \frac{e^{\pi_{i,t}}}{\sum_{j=1}^{K} e^{\pi_{j,t}}}$. Suppose that at time step $t$, the arm $i$ has been drawn, obtaining a reward $r_t$. Then the preference is updated as $\pi_{i,t+1} = \pi_{i,t} + \beta(r_t - \bar{r}_t)$, where $\bar{r}_t$ is the average expected reward. At the end of each time step $t$, $\bar{r}_{t+1}$ is computed as $\bar{r}_{t+1} = (1 - \alpha)\bar{r}_t + \alpha r_t$.

As announced above, this algorithm does not fit our model. In a federated learning context with $K$ data owners, additionally with security concerns, we consider that $p_{i,t}$ is sensible and must be kept locally in DO$_i$. To compute $p_{i,t+1}$, each DO$_i$ must know his preference $\pi_{i,t+1}$ which subsequently require $\bar{r}_t$, implies that each DO$_i$ must keep updated $\bar{r}_t$. The fact is, from $\bar{r}_t$ and $\bar{r}_{t+1}$, we are able to retrieve $r_t$ as follows:

$$\bar{r}_{t+1} - (1 - \alpha)\bar{r}_t = \frac{(1 - \alpha)\bar{r}_t + \alpha r_t - (1 - \alpha)\bar{r}_t}{\alpha} = r_t$$

Whatever the way used to compute $\bar{r}_{t+1}$ in a federated learning context while ensuring a confidentiality remains difficult with this algorithm. The idea behind the incompatibility of SAMBA and the Reinforcement Comparison is that some collusion between two DO$_i$ nodes are required to compute local information, in this case $p_{i,t}$.

C.4 Proof of Genericity

We recall the following theorem

**Theorem 1** Let $\mathcal{A}$ a standard multi-armed bandits and $\mathcal{A}'$ the distributed version of $\mathcal{A}$, we state that if $\mathcal{A}'$ respecting the properties cited above, then $\mathcal{A}'$ can be simulated by SAMBA.

**Proof.** To prove the theorem, we do a proof by recurrence over the number of computation rounds $nbIteration_{\mathcal{A}}$. 

def SelectArchitecture(t, l, A)  
    return Informed

def ComputeScore(t, l, s_i, n_i, A)  
    sample \( \theta_i \sim \text{Beta}(s_i + 1, n_i - s_i + 1) \)  
    return \( \theta_i \)

def SelectArm(t, l, \( \sigma_t([\alpha_t v_{i,t}]_{i\in[K]}), A \))  
    return \( \arg \max \sigma_t([\alpha_t v_{i,t}]_{i\in[K]} \))

def HandleSelect(t, l, b_i, s_i, n_i, A)  
    if \( b_i = 1 \)  
        \( s_i \leftarrow s_i + \text{pull}(i) \)  
        \( n_i \leftarrow n_i + 1 \)  
    return \( (s_i, n_i) \)

(a) The Thompson Sampling Algorithm

def HandleSelect(t, l, b_i, s_i, n_i, A)  
    if \( l = 1 \)  
        (\( \beta \)) \leftarrow \text{Parameter from } A  
        if \( b_i = 1 \)  
            \( p_{i,t+1} = p_{i,t} + \beta(1 - p_{i,t}) \)  
        else  
            \( p_{i,t+1} = p_{i,t} + \beta(0 - p_{i,t}) \)  
        \( A \leftarrow (\beta, p_{i,t+1}) \)  
    else  
        if \( b_i = 1 \)  
            \( s_i \leftarrow s_i + \text{pull}(i) \)  
            \( n_i \leftarrow n_i + 1 \)  
    return \( (s_i, n_i) \)

(b) The Pursuit algorithm

![Fig. 12. Thompson Sampling and Pursuit algorithms](image)

- **The Initialisation case** occurs when \( n_{\text{iteration}}_A = 1 \). It means that only one function \( f \) is needed by the central server. By assumption, \( A' \) is composed by a central server which at each time step \( t \), received \( K \) values produced by each arm, and selects the arm \( i_m \) to be pulled. We split the central server in two independent sub-servers called respectively Comp and Controller, each DO_i will send its value to the Controller. Once all values have been received by Controller, it constructs a list of them, permuted the created list with a pseudo-random permutation \( \sigma_t \), and send it to Comp. Once that Comp has received the permuted list, it applies the function \( f \) over the \( K \) elements in the received list in order to get the selected arm \( i_m \). After that, it sends to Controller the selection bits list \( \sigma_t(\{i_{i=1}^m\}_{i\in[K]}) \). Controller apply the inverse permutation function on the selection bits list, and distribute selection bits. As Controller is limited to play like a proxy, with the task to compute a permutation, hence it does not need to see data in clear. Hence, we create a secure communication channel between each DO_i and Controller, thanks to symmetric encryption. Moreover, it would be preferable to not see data in clear but masked, in case where sensitive information is sent by a DO_i. Hence, we use a same mask generated by each DO_i, thanks to the seed \( seed_{s_i} \), assumed transmitted at the beginning of the algorithm. This is assumed respecting the order-preserving and proportion-preserving mask. Thus, we can modify \( A' \) in order to be simulated by SAMBA.

- **By Inheritance on \( n_{\text{iteration}}_A \),** we suppose that the theorem holds when there are \( n_{\text{iteration}}_A \) functions \( f \), denoted \( n_{\text{iteration}}_A = m \) for simplicity, which leads to denote the succession of arm selection functions \( f_1, ..., f_m \). By recurrence on \( n_{\text{iteration}}_A \), we prove that the theorem still holds when we add a new function \( f_{m+1} \), which leads to have \( f_1, ..., f_m, f_{m+1} \). By hypothesis, we assume that each one of these function depends only on time step \( t \), computation round \( l \) and \( K \) values produced by each DO_i at given time step \( t \) and computation round \( l \). By adding
/* Parameters Setup */
receive $N, A, seed_{\alpha_i}$ from Controller

$nbl_{\text{iteration}_{A}} \leftarrow$ number of iterations needed by $A$

$(t, l, b_{i}, t, s_{i}, n_{i}) \leftarrow (i, 1, 1, 0, 0)$

$(s_{i}, n_{i}) \leftarrow \text{HandleSelect}(t, l, b_{i}, t, s_{i}, n_{i}, A)$

/* Core of the protocol */
for $t \in [K+1, N]$
    for $l \in [nbl_{\text{iteration}_{A}}]$
        archi $\leftarrow \text{SelectArchitecture}(t, l, A)$
        if archi = Informed
            $\alpha_{t} \leftarrow \text{nextValueFromSeed}(seed_{\alpha_{t}}, t, l)$
            $v_{i, t} \leftarrow \text{ComputeScore}(t, l, s_{i}, n_{i}, A)$
            send $\text{Enc}(\alpha_{t} \cdot v_{i, t})$ to Controller
        receive $\text{Enc}(b_{i, t})$ from Controller
        $b_{i, t} \leftarrow \text{Dec}(\text{Enc}(b_{i, t}))$
        $(s_{i}, n_{i}) \leftarrow \text{HandleSelect}(t, l, b_{i}, t, s_{i}, n_{i}, A)$

/* Cumulative Reward Computing */
send $\mathcal{E}(s_{i})$ to Controller

(a) Pseudo-code of SAMBA for $DO_{i}$

/* Parameters Setup */
receive $N, A$ from DC
send $N, A$ to the Comp
$seed_{\alpha_{t}} \leftarrow$ new seed
for $i \in [K]$
    send $N, A, seed_{\alpha_{t}}$ for each $DO_{i}$

$nbl_{\text{iteration}_{A}} \leftarrow$ number of iterations needed by $A$

/* Core of the protocol */
for $t \in [K+1, N]$
    for $l \in [nbl_{\text{iteration}_{A}}]$
        archi $\leftarrow \text{SelectArchitecture}(t, l, A)$
        if archi = Informed
            for $i \in [K]$
                receive $\text{Enc}(\alpha_{t} \cdot v_{i, t})$ from $DO_{i}$
                $\sigma_{t} \leftarrow$ new permutation
                send $\sigma_{t} \cdot (\text{Enc}(\alpha_{t} \cdot v_{i, t}))$ to Comp
            receive $\sigma_{t} \cdot (\text{Enc}(b_{i, t}))$ from Comp
            for $i \in [K]$
                send $\sigma_{t} \cdot (\text{Enc}(b_{i, t}))$ to $DO_{i}$

/* Cumulative Reward Computing */
for $i \in [K]$
    receive $\mathcal{E}(s_{i})$ from $DO_{i}$
    send $\mathcal{E}(s_{1}) \cdot ... \cdot \mathcal{E}(s_{K})$ to the DC

(b) Pseudo-code of SAMBA for Controller

/* Parameters Setup */
receive $N, A$ from Controller

$nbl_{\text{iteration}_{A}} \leftarrow$ number of iterations needed by $A$

/* Core of the protocol */
for $t \in [K+1, N]$
    for $l \in [nbl_{\text{iteration}_{A}}]$
        archi $\leftarrow \text{SelectArchitecture}(t, l, A)$
        if archi = Random
            $i_{m} \leftarrow$ random integer $i_{m} \in [K]$
            send $\{\text{Enc}(\Pi_{i=i_{m}})\}_{i \in [K]}$ to Controller
        else
            receive $\sigma_{t}(\{\text{Enc}(\alpha_{t} \cdot v_{i, t})\}_{i \in [K]})$ from Controller
            $\mathcal{V} \leftarrow \sigma_{t}(\{\text{Dec}(\text{Enc}(\alpha_{t} \cdot v_{i, t}))\}_{i \in [K]})$
            $\sigma_{t}(i_{m}) \leftarrow \text{SelectArm}(t, l, \mathcal{V}, A)$
            send $\sigma_{t}(\{\text{Enc}(\Pi_{i=i_{m}})\})$ to Controller

(c) Pseudo-code of SAMBA for Comp

Fig. 13. Pseudo-code of the extended protocol. The pseudo-code of DC does not differ from the base version of SAMBA.
the function \( f_{m+1} \), we increment the number of computation round and each \( \text{DO}_i \) will produce a new value at computation round \( m+1 \). Hence, \( f_{m+1} \) does not depend on values computed during previous computation rounds, then adding a new computation rounds does not break our properties, implying that \( A' \) can be be simulated by SAMBA. By recurrence, our theorem is proved.

\[ \square \]

**APPENDIX D**

**EXPERIMENTS**

We detail in this section how we chosen parameters provided to some algorithms. Moreover, we expose other experiments done on Jester database.

**D.1 Fixing Parameters**

The \( \varepsilon \)-greedy Softmax, Pursuit and need some parameters to work. We must be aware about the parameters set up, because of significant different configuration can lead to a radically modified behavior, deeper explained in the section \[ D.2 \] So far, we decide to implement a strategy to fit the most optimal parameters, by including an automatic parameters function which try several configuration on a given dataset, which and returns the optimal parameters, based on the yield rewards.

In particular, we have done experiments on MovieLens dataset and Jester dataset, with both 100 arms. Then, for each dataset, we perform the parameters tuning as follows:

- For \( \varepsilon \)-greedy, we run the algorithm with \( \varepsilon \in \{0, 0.1, 0.2, ..., 1\} \). For both MovieLens and Jester, the best obtained configuration is \( \varepsilon = 0.1 \).
- For Softmax, we run the algorithm with \( \tau \in \{0.0, 0.01, 0.02, ..., 0.1\} \). For MovieLens, we obtained \( \tau = 0.06 \), while \( \tau = 0.02 \) for the Jester dataset.
- For Pursuit, we run the algorithm with \( \beta \in \{0.0, 0.01, 0.02, ..., 0.1\} \). For MovieLens, we obtained \( \beta = 0.1 \), while \( \beta = 0.2 \) for the Jester dataset.

**D.2 Impact of parameters on algorithm**

**D.2.1 \( \varepsilon \)-greedy**

The \( \varepsilon \)-greedy algorithm needs to set an \( \varepsilon \in [0, 1] \), used in the exploitation/exploration trade-off. In case where \( \varepsilon \) tends to zero, \( \varepsilon \)-greedy acts like a pure greedy algorithm. At the opposite when \( \varepsilon \) tends to 1, \( \varepsilon \)-greedy choose randomly an arm uniformly.

**D.2.2 Softmax**

The Softmax algorithm requires the \( \tau \) parameter, with \( \tau \in \mathbb{R} \), used to compute the score. When \( \tau = 0 \), Softmax acts like pure greedy. As \( \tau \) tends to infinity, the algorithms picks arms uniformly at random.

**D.2.3 Pursuit**

The Pursuit algorithm uses a \( \beta \) learning rate, with \( \beta \in [0, 1] \), to update the probability of an arm \( i \in [K] \) to be drawn. To set the learning rate value, we must deal with the convergence and learning speed trade-off. If the learning rate is too much lower, the algorithm will take a long time to converge, while a too much higher learning rate will conduct to a bad convergence. In our experiment, the learning rate does not heavily affect the behavior of the algorithm, except when \( \beta \) is 0, where Pursuit selects a random arm at each turn.

**D.3 Experiments results on Jester**

The two first plots show the obtained rewards for each algorithm, when \( N \) increase with \( K = 100 \), and when \( K \) increase with \( N = 100000 \). The two last plots show the scalability with respect to \( N \) with \( K = 100 \), and the scalability with respect to \( K \) with \( N = 100000 \). The last plot is a zoom on the impact of each component on the execution time with \( K = 10 \) and \( N = 100000 \). The dotted lines are the standard algorithms and the full lines are their SAMBA versions.