# DIPSAUCE: Efficient Private Stream Aggregation Without Trusted Parties

Joakim Brorsson[⋆] and Martin Gunnarsson[⋆⋆]

Lund University {joakim.brorsson,martin.gunnarsson}@eit.lth.se

**Abstract.** Private Stream Aggregation (PSA) schemes are efficient protocols for distributed data analytics. In a PSA scheme, a set of data producers can encrypt data for a central party so that it learns the sum of all encrypted values, but nothing about each individual value. Thus, a trusted aggregator is avoided. However, all known PSA schemes still require a trusted party for key generation. In this paper we propose the first PSA scheme that does not rely on a trusted party. We argue its security against static and mobile malicious adversaries, and show its efficiency by implementing both our scheme and the previous state-of-the-art on realistic IoT devices, and compare their performance. Our security and efficiency evaluations show that it is indeed possible to construct an efficient PSA scheme without a trusted central party. Surprisingly, our results also show that, as side effect, our method for distributing the setup procedure also makes the encryption procedure *more efficient* than the state of the art PSA schemes which rely on trusted parties.

**Keywords:** Private Stream Aggregation · IoT · Privacy.

## 1 Introduction

Internet of Things (IoT) data analytics enable central parties to learn statistics derived from device data. This data is often privacy sensitive, and thus systems must be designed with privacy in mind. Consider for example the concept of smart metering [27] where a central party calculates the sum of readings of household electricity meters in real-time. Disclosing individual readings in real-time reveals a surprisingly high amount of privacy sensitive data about a household [34]. Thus an untrusted central party should not have access to individual data readings. There exist works studying how to centrally derive statistics without revealing individual data points for the case of smart meters [28,31,23]. We are however interested in developing general techniques for IoT data analytics.

A known technique for data analytics is *Functional Encryption* (FE) [8], where knowledge of a *functional decryption key* allows function evaluation on

⋆ Supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

⋆⋆ Supported by grant RIT17-0032 from the Swedish Foundation for Strategic Research.

encrypted data. For IoT data analytics on privacy sensitive data, the FE subclass of (Decentralized) Multi Client Functional Encryption ((D)MCFE) is particularly interesting, since it defines FE for multiple parties contributing encrypted data for a centralized evaluator. However, IoT devices are often *constrained* [9], *i.e.* they have low computational power and memory, operate over low throughput lossy networks or are battery powered. Even the most efficient DMCFE schemes [18,2,19], which evaluate inner products of encrypted data, are too costly for constrained environments since they rely on bilinear parings or have ciphertext sizes proportional to the number of data producers.

When the evaluated function is specifically a *sum*, one can instead consider Secure Aggregation (SA) [7] and Private Stream Aggregation (PSA) [39].

SA schemes, which are proposed in the context of federated learning, compute the plaintext sum of a set of encrypted vectors, with a focus on *robustness* against frequent client drop-outs (*e.g.* 6-10% drop-outs per summation [6]). The robustness is achieved by introducing *multiple rounds of client interaction and computation* per summation, making SA schemes unfit for constrained devices.

PSA schemes have instead been suggested for IoT data analytics applications which involve constrained devices. PSA schemes also compute the plaintext sum of encrypted values, but instead focus on *efficiency*. As such, they use efficient primitives and avoid client interaction. However, to the best of our knowledge, all known PSA schemes rely upon a *trusted party* during the setup procedure, which includes key generation [39,15,26,30,5,21,3,42,22,44,41].

We argue that since the purpose of a PSA scheme is to allow an untrusted party to derive statistics without learning anything about individual data points, relying on a trusted party is not in line with the goals of PSA. Such a design erodes trust in a privacy enhancing technology and is particularly engraving for PSA schemes, since their purpose is to avoid a central party with access to individual data. We therefore propose DIPSAUCE, a PSA scheme which does not rely on trusted parties, and which is suitable for constrained devices.

## 1.1 Contributions

In this paper we (1) introduce a definition for *distributed setup* PSA and its corresponding security model, (2) present DIPSAUCE, the first PSA scheme which does not rely on a trusted party, (3) prove this scheme secure under static corruptions, (4) describe modifications for security under mobile corruptions, (5) demonstrate its efficiency by implementing it on realistic, off-the-shelf devices advertised as being suitable for *e.g.* smart-metering. Since no other PSA scheme is evaluated on realistic devices, we also (6) implement two state-of-the-art PSA schemes [22,44] on the same devices and compare the performance to our scheme. All code and raw data are made publicly available [12,13].

Looking ahead, DIPSAUCE shows a speedup of 78x and 49x respectively compared with the suggestions for a distributed setup in KH-PRF-PSA [22] and LaSS-PSA [44] for 10000 parties. For the encryption procedure our results show a speedup of 22x compared to KH-PRF-PSA and 50x compared to LaSS-PSA.

### 1.2 Our Techniques

It is known how to distribute the setup procedure between all $n$ parties (see Section 1.3 for details). This is however too costly for constrained devices. Our key innovation is a mechanism which reduces the number of key agreements to $k << n$, while still tolerating a high degree ($>> k$) of corruptions among the parties, and without introducing network overhead. We do this by leveraging a *$k$-regular graph* of order $n$ where each vertex represents a party in the system. The graph is randomly permuted, and each party is assigned a committee, consisting of the parties represented by its $k$ neighbouring vertices in the randomly permuted graph. Each party then engages in non-interactive pairwise key exchange, *but only has to do so for its committee of the $k$ random neighbours.*

By using a random permutation of a $k$-regular graph, we guarantee a random committee of the correct size for each user. This further enables us to let the *graph structure* (but not the random permutation of it) be known in advance and stored locally with the parties. Each party can then, instead of expensively obtaining a large random graph over the network, locally derive a random permutation of the graph defined by a single shared randomness seed from an external distributed randomness beacon service [17,20], This results in minimal network overhead which enables a distributed setup on constrained devices.

### 1.3 Related Work

The current state-of-the-art for PSA schemes are the KH-PRF-PSA [22] and LaSS-PSA [44] schemes. While TERSE [41] measures faster encryption times, these results are not directly comparable with KH-PRF-PSA and LaSS-PSA since they are based on precomputations and only measure the "on-line" time. Similar precomputations can be done for LaSS-PSA and KH-PRF-PSA as well, and the resulting "on-line" stages then consists of a single modular addition, while the TERSE "on-line" stage uses more complex operations. A direct comparison is therefore needed before it can challenge the state-of-the-art.

Notably, both KH-PRF-PSA and LaSS-PSA briefly discuss how to avoid a trusted party by using a distributed setup. Both works propose to adopt the methods of the DMCFE scheme by Chotard et al. [19], where centrally generated keys are replaced with pairwise agreed upon keys between all $n$ parties. These methods are secure under *adaptive corruptions*. However, neither KH-PRF-PSA nor LaSS-PSA have any formal protocol description, security evaluation or efficiency evaluation of the proposal for a distriburted setup. As we show in Section 4, these methods are too inefficient for constrained environments.

We also note that the approach of securing a distributed setup by pairwise user key agreement is present in SA schemes [7], and that Bell et al. [4] propose a version of the technique which lessens the CPU load by establishing smaller random committees. However, in contrast to our non-interactively generated random permutation of a $k$-regular graph, Bell et al. resort to users *interactively generating a directed random graph* over the network to provide security against static malicious adversaries. Therefore, their approach does not transfer to PSA schemes, which need to work with constrained network resources. Bell et al. do also propose a version with non-interactive graph generation, but which can only achieve security against a *semi-honest static* adversary.

Let us summarize. State-of-the-art PSA schemes [44,22] sketch distributed setup procedures for PSA schemes, which are possible to prove secure under *adaptive* corruptions, but which are infeasibly inefficient due to high CPU overhead. The state-of-the-art SA scheme [7] also provides a distributed setup, proved secure under *static malicious* corruptions, but which is infeasible for constrained devices due to high network overhead.

**Related Concurrent Work** Concurrently to our work, the FLAMINGO SA scheme [32] has proposed to rely on a similar mechanism for non-interactively establishing small random committees. Let us elaborate on the differences between DIPSAUCE and FLAMINGO. DIPSAUCE is a PSA scheme focused on efficiency and suitable for constrained devices. FLAMINGO is an SA scheme focused on dropout resilience and not suitable for constrained devices. Both works rely on a novel strategy where a randomness beacon is used to non-interactively construct a graph with small random committees of neighbours. FLAMINGO establishes a graph by joining 2 of the $n$ vertices with an edge if a random value is below a threshold. DIPSAUCE establishes a graph by permuting a $k$-regular graph based on a random input. In FLAMINGO, the number of neighbours to a vertex is probabilistic, while in DIPSAUCE each vertex always has $k$ neighbours, which allows a simpler security proof. Our work first appeared on ePrint at the 17:th of February 2023 and [32] later appeared on the 4:th of March 2023. Although we published our work first, to the best of the author's knowledge both works were developed unaware of each other.

## 2 Preliminaries

*Notation:* $\lambda \in \mathbb{N}$ denotes the computational security parameter which controls the security level of cryptographic components. A specific party in a scheme is denoted $\mathcal{P}_i$. We use the notation $\vec{a}[i]$ to denote the $i$'th element of the vector $\vec{a}$. We use $[n]$ as a short hand notation for $\{1, \ldots, n\}$. Let $\mathsf{Perm}(n)$ be the lexicographically ordered set of permutations of $[n]$. We denote the $k$:th permutation of this set as $\mathsf{Perm}_k(n)$. For any permutation of $[n]$, $\rho_k = \mathsf{Perm}_k(n)$, we denote the value of $i$:th element in $\rho_k$ as $\rho_k(i)$. We denote a graph as $G = (V, E)$, where $V$ is the set of vertices in the graph and $E$ the set of edges. The set of neighbouring vertices of $v_i \in V$ is denoted $N(v_i)$, and $\vec{J_i}$ denotes the set of all indices of vertices in $N(v_i)$. We denote the floor function of $x$, *i.e.* the greatest integer less than or equal to $x$, as $\lfloor x \rfloor$. As a shorthand we sometimes write $(-1)^{(i<j)}$. In this notation $(i < j)$ is the boolean function so that $(-1)^{(i<j)} = (-1)$ when $i < j$ and $(-1)^{(i<j)} = 1$ when $i > j$. The function is undefined for $i = j$.

*Private Stream Aggregation:* We here give an informal definition of *standard* PSA. A formal definition is available in the full version of this paper [11]. Our notion of *distributed setup* PSA is given in Section 3.

In a Private Stream Aggregation scheme $\mathsf{PSA} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Aggr})$ an aggregator can learn the sum of the inputs $\{m_1, \dots, m_n\}$, from a set of parties $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ without learning the individual inputs. The $\mathsf{Setup}$ procedure is executed by a trusted party and generates an encryption key for each party, and the aggregation key for the aggregator. The $\mathsf{Enc}$ procedure is executed by party $\mathcal{P}_i$, and encrypts input $m_i$. Then, the aggregator can execute $\mathsf{Aggr}$ which outputs the sum of all user inputs. Informally, a PSA scheme is *correct* if the output of $\mathsf{Aggr}$ will always be equal to the sum of the inputs, and *secure* if nothing but the sum of the inputs of *honest* users is learned by an adversary.

*$k$-Regular Graphs:* A $k$-regular graph is a graph in which each vertex has exactly $k$ neighbours. It is well known how to efficiently generate regular graphs [33].

*Distributed Randomness Beacons:* In a Distributed Randomness Beacon (DRB) protocol [17], a set of entropy providers jointly compute *publicly verifiable randomness*. The beacon function, $r = \mathsf{Beacon}(t)$, returns an $m$-bit near-uniformly random value $r$ at each epoch $e$. Any party can obtain and verify this randomness, *i.e.* also external parties not part of the randomness generation.

Informally, a secure DRB should be *unpredictable*, *i.e.* the advantage for an adversary predicting $r$ before the epoch $e$ begins should be negligible, *unbiased*, *i.e.* $r$ must be statistically close to an $m$-bit uniformly random string, and *live*, *i.e.* the probability of no output during each epoch should be negligible. These properties should hold also when a fraction of the entropy providers are corrupt.

*Non-Interactive Key Exchange:* A $\mathsf{NIKE}$ scheme, defined in Definition 1, is correct if $\Pr[\mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_j) = \mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_j, \mathsf{sk}_i)] = 1$. A $\mathsf{NIKE}$ scheme is secure against a computationally bounded adversary given $(\mathsf{pp}, \mathsf{pk}_i, \mathsf{pk}_j)$ if it cannot distinguish the output of $\mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_j)$ from a random string of the same length. We refer to [19] for a full definition of the security game.

**Definition 1 (NIKE).** *A Non-Interactive Key Exchange scheme establishes a shared key between two parties and consists of the following algorithms:*
$\mathsf{Setup}(\lambda)$*: On input a security parameter $\lambda$, output public parameters* $\mathsf{pp}$.
$\mathsf{KeyGen}(\mathsf{pp})$*: On input the public parameters* $\mathsf{pp}$*, output a keypair* $(\mathsf{pk}_i, \mathsf{sk}_i)$.
$\mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_j)$*: On input the public parameters* $\mathsf{pp}$*, a public key* $\mathsf{pk}_i$ *and secret key* $\mathsf{sk}_j$*, deterministically output a shared key $K$.*

*Pseudo-Random Functions:* Let $\mathcal{F}$ denote a family of efficiently-computable functions $F_k : X \to Y$ indexed by $k \in K$. The family $\mathcal{F}$ is said to be a $(t, \epsilon)$ strong $\mathsf{PRF}$ if for every $k \in K$, no adversary $\mathcal{A}$ running in time $t$ can distinguish $F_k$ from a random function $f : X \to Y$. We will denote such a function $F_k$ as $\mathsf{PRF}_k$. Further, $\mathcal{F}$ is *additively* key-homomorphic if $\forall F_{k_i}, F_{k_j} \in \mathcal{F}$, the condition $F_{k_i}(x) + F_{k_j}(x) = F_{k_i+k_j}(x)$ holds. We denote such a function as $\mathsf{KH\text{-}PRF}_k$.

*Sum-of-PRFs:* The sum-of-PRFs technique, first introduced in [16], allows parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ to derive the sum of their inputs $\{m_1, \ldots, m_n\}$ without revealing the individual $m_i$:s from honest users. An adversary who corrupts the aggregator and $m < n-2$ parties can then only learn the *sum* of the inputs of the honest users. The technique assumes that each pair of users, $\mathcal{P}_i, \mathcal{P}_j$ has a shared secret $K_{i,j}$. To mask its message $m_i$, $\mathcal{P}_i$ derives $c_i \leftarrow m_i + \sum_{j \in [n] \setminus \{i\}} (-1)^{i<j} \cdot \mathsf{PRF}_{K_{i,j}}(x)$ (note the $(-1)^{i<j}$ notation). Then, the sum of all $m_i$ can be calculated as $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} m_i$. Summing any set smaller than $n$ of $c_i$ containing at least 2 ciphertexts from honest users will result in a random output.

## 3 DIPSAUCE

As we show in Section 4, the suggestions for distributing the setup in state-of-the-art PSA schemes [44,22] are too inefficient for use on constrained devices. To address this, we now present our protocol for DIstributed setup PSA for Use in Constrained Environments (DIPSAUCE). It takes inspiration from the LaSS-PSA scheme [44, Section 4], but crucially differs by not relying on a trusted party.

*Approach:* The suggestions for distributing the setup procedures of LaSS-PSA and KH-PRF-PSA use the sum-of-PRFs technique, which works by each party evaluating a PRF once for each party in its *committee*. This committee consists of *all other parties*, and thus its size is $n-1$. In these schemes, a party is secure against an *adaptive* adversary which corrupts up to $n-2$ of the committee parties (but not the targeted party itself). While this is a very strong security guarantee, the resulting protocol is rendered too inefficient for practical use (see Section 4). The main bottleneck for this inefficiency is the committee size.

Simply shrinking the committee size would make the protocol more efficient, but simultaneously lower the corruption tolerance, sacrificing security. How then to shrink the committee size without also lowering the corruption tolerance? A key insight is that a *static* or *mobile* adversary cannot target devices in a committee for corruption (within an epoch) if it cannot predict what devices constitutes the committee. Using an unpredictable committee of size $k << n$ we can create a more efficient construction, secure in the presence of a *static* or a *mobile* adversary capable of corrupting up to $t$ devices, where $k < t < n$.

The technical novelty of our protocol lays in how it uses a $k$-regular graph and a randomness beacon to non-interactively and efficiently establish unpredictable committees. The protocol defines each committee using the output of an external distributed public randomness beacon. However, an efficient protocol cannot directly use the beacon output to determine the committees. Sampling $n$ committees of size $k$ and sending this data to the devices would mean sending $\mathcal{O}(nk)$ group elements to each device, which is not feasible for constrained devices or networks. Instead, we first let each device be represented as a vertex in a $k$-regular graph which is part of the system configuration. Then, a *single* output of the beacon is used to determine a pseudorandom permutation of this graph. The committee of each party is then determined by the $k$ neighbours in the randomly permuted graph. This committee is then used in a *threshold* sum-of-PRFs where each party evaluates a PRF for only $k$ other parties.

*Aggregation output:* In line with previous PSA schemes, we consider a definition for PSA which outputs the sum of all plaintexts to the aggregator, *i.e.* we do not strive to achieve differential privacy. In contrast to existing definitions of PSA, no secret key is needed to aggregate the sum of plaintexts. This is a more general definition. If it is a desired system property to allow only one specific party to aggregate, then this property can be obtained by sending the ciphertexts over an encrypted channel to the aggregator, or by including the aggregator among the encrypting parties and letting it encrypt zero without publishing the ciphertext.

## 3.1 Syntax and Security Model

*Assumptions:* We assume that all parties have access to a distributed randomness beacon and a Public Key Infrastructure (PKI) assumed to behave correctly, *e.g.* not accepting duplicate keys and verifying knowledge of private keys, etc. While such a PKI is a standard assumption, we note that it is possible to *distributively audit* a PKI for correct behaviour [29]. We also assume that each vertex in $G$ has been assigned an index.

*Corruptions:* We consider an adversary $\mathcal{A}$ capable of corrupting any party $\mathcal{P}_i$, up to a threshold of $t$ parties. Once a party is corrupt, $\mathcal{A}$ takes control of the execution of that party, meaning that it controls the actions and learns the internal state throughout the execution. The set of corrupt parties is denoted $\mathcal{C}$.

**Definition 2 (Distributed Setup Private Stream Aggregation).** *A Distributed Setup Private Stream Aggregation (DS-PSA) scheme over $\mathbb{Z}_R$, where $R \in \mathbb{N}$, is defined for a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and a special party called the evaluator $\mathcal{E}$, and consists of the following procedures:*

- $\mathsf{Setup}(\lambda, conf)$*: On input a security parameter $\lambda$ and optional configuration parameters $conf$, the procedure outputs the system parameters* $\mathsf{pp}$.
- $\mathsf{KeyGen}(\mathsf{pp}, i)$ *On input the system parameters* $\mathsf{pp}$ *and the users index in the system, $i$, output an encryption key* $\mathsf{ek}_i$.
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, m_i, l)$*: On input the system parameters* $\mathsf{pp}$*, an encryption key* $\mathsf{ek}_i$*, a message $m_i$ and a label $l$, output an encryption $c_i$ of $m_i$ under* $\mathsf{ek}_i$.
- $\mathsf{Aggr}(\mathsf{pp}, \{c_i\}_{i\in[n]}, l)$*: On input the system parameters* $\mathsf{pp}$*, a set of $n$ ciphertexts $\{c_i\}_{i\in[n]}$ and a label $l$, output the sum of all plaintexts, $M \pmod R$.*

Note that, as is often the case in PSA, our scheme returns the sum of the encrypted values modulo $R$, where $R$ is a system parameter.

We say that a Distributed Setup PSA scheme is *correct* if for all $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda, conf)$, $m_i$, $l$, $\{\mathsf{ek}_i \leftarrow \mathsf{KeyGen}(\mathsf{pp}, i)\}_{i\in[n]}$, we have:

$$\Pr\left[\mathsf{Aggr}\left(\{\mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, m_i, l)\}_{i\in[n]}\right) = \sum_{i=1}^{n} m_i\right] = 1$$

```
$\mathsf{AO}_b(\lambda, n, conf, \mathcal{A})$

$L \leftarrow \emptyset$
$\mathcal{C} \leftarrow \mathcal{A}, s.t. |\mathcal{C}| \leq t$
$\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda, conf)$
for $i$ where $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{C}$ do
    $\mathsf{ek}_i \leftarrow \mathsf{KeyGen}(\mathsf{pp}, i)$
end for
$\gamma \leftarrow \mathcal{A}^{\mathsf{QEnc},\mathsf{QLeftRight}}$
return $\gamma \overset{?}{=} b$
```

Fig. 1: The AO experiment defining security for a distributed setup PSA scheme.

A DS-PSA scheme is secure if an adversary has a negligible probability of winning the game for Aggregator Obliviousness (AO) in Definition 3.

**Definition 3 (Aggregator Obliviousness (AO)).** *Security is defined via the game of Aggregator Obliviousness* $\mathsf{AO}_b(\lambda, n, \mathcal{A})$*,* $b \in \{0, 1\}$ *in Figure 1.* $\mathcal{A}$ *denotes the adversary with access to the following oracles:*

- $\mathsf{QEnc}(i, m_i, l^*)$*: Given a user index* $i$*, a message* $m_i$ *and a label* $l^*$*, if* $(i, l^*) \notin L$ *and* $\mathcal{P}_i \notin \mathcal{C}$ *then it lets* $L \leftarrow L \cup \{(i, l^*)\}$ *and returns* $c_i = \mathsf{Enc}(\mathsf{ek}_i, m_i, l^*)$*.*
- $\mathsf{QLeftRight}(\mathcal{U}, \{m_i^0\}_{i \in \mathcal{U}}, \{m_i^1\}_{i \in \mathcal{U}}, l^*)$*: Given a set* $\mathcal{U}$ *of user indices, two sets* $\{m_i^0\}_{i \in \mathcal{U}}$ *and* $\{m_i^1\}_{i \in \mathcal{U}}$*, and a label* $l^*$*, it checks if* $\forall i \in \mathcal{U} : (i, l^*) \notin L$ *and* $\{\mathcal{P}_i\}_{i \in \mathcal{U}} \cap \mathcal{C} = \emptyset$ *and no previous calls has been made to* $\mathsf{QLeftRight}$*. If further* $\{\mathcal{P}_i\}_{i \in \mathcal{U}} \cup \mathcal{C} = \{\mathcal{P}_i\}_{i \in [n]}$ *it also checks if* $\sum_{i \in \mathcal{U}} m_i^0 = \sum_{i \in \mathcal{U}} m_i^1$*. If all checks return true, it lets* $L \leftarrow L \cup \{(i, l^*)\}_{i \in \mathcal{U}}$ *and returns* $\{c_i\}_{i \in \mathcal{U}}$*, where* $c_i = \mathsf{Enc}(\mathsf{ek}_i, m_i^b, l^*)$*.*

*At the end of the game,* $\mathcal{A}$ *outputs a guess,* $\gamma$*, of whether* $b$ *equals 0 or 1.*

Static corruptions is modeled by the adversary picking the set $\mathcal{C}$ of at most $t$ corrupt parties at the start of the game. We model *encrypt-once* security, i.e. restricting each party to only encrypt a single message per label (which is the natural usage of the scheme), by both QEnc and QLeftRight maintaining the set $L$, where they store which label has been used for each user and ignoring any requests which reuse labels. Further, since any party has the ability to aggregate in Definition 2, the QLeftRight enforces that $\sum_{i \in \mathcal{U}} m_i^0 = \sum_{i \in \mathcal{U}} m_i^1$ when all honest users are part of the QLeftRight call. This prevents $\mathcal{A}$ from trivially winning the game by receiving a ciphertext for each honest user and then checking whether the output of Aggr contains $\{m_i^0\}_{i \in \mathcal{U}}$ or $\{m_i^1\}_{i \in \mathcal{U}}$.

This AO-game is similar to the AO-games for LaSS-PSA and KH-PRF-PSA. The main differences are the modeling of corruptions as full party takeovers rather than a key leaking oracle, and the lack of a dedicated aggregator key.

**Protocol 1** – DIPSAUCE

```
Setup($\lambda, conf = \{n, k, time, R\}$):
 1: Generate a $k$-regular graph $G = (V, E)$ where $|G| = n$
 2: $\mathsf{npp} \leftarrow \mathsf{NIKE.Setup}(\lambda)$
 3: return $\mathsf{pp} = \{\mathsf{npp}, n, k, G, time, R\}$

KeyGen($\mathsf{pp}, i$):
 1: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{NIKE.KeyGen}(\mathsf{npp})$
 2: Post $(\mathcal{P}_i, \mathsf{pk}_i)$ to the PKI
 3: $r \leftarrow \mathsf{Beacon}(time)$
 4: $\rho \leftarrow \mathsf{Perm}_r(n)$
 5: Let $\vec{J_i}$ be the vector s.t $\forall \vec{J_i}[\ell] = j : v_j \in N(v_{\rho(i)})$, (i.e. the indices of $\mathcal{P}_i$:s neighbors
    in the permuted graph)
 6: for $\ell \in \{1, \ldots, k\}$ do
 7:     $\ell' = \vec{J_i}[\ell]$
 8:     Wait until the PKI returns an entry $\mathsf{pk}_{\ell'}$ for $\mathcal{P}_{\ell'}$
 9:     $\vec{K_i}[\ell'] \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_{\ell'}, \mathsf{sk}_i)$
10: end for
11: return $\mathsf{ek}_i = (\vec{K_i}, \vec{J_i})$

Enc($\mathsf{pp}, \mathsf{ek}_i = (\vec{K_i}, \vec{J_i}), m_i, l$):
 1: $t_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i < \vec{J_i}[\ell]} \cdot \mathsf{PRF}_{\vec{K_i}[\ell]}(l)$
 2: return $c_i = (t_i + m_i) \pmod R$

Aggr($\mathsf{pp}, \{c_i\}_{i \in [n]}$):
 1: return $M = \sum_{i \in n} c_i \pmod R$
```

## 3.2 Construction

The protocol is defined in Protocol 1. It is run with $n$ parties, assigned indexes from 1 to $n$ in an arbitrary fashion (*e.g.* based on network addresses).

First, the Setup procedure must be executed, and the public parameters distributed to each party. Then, each party can compute its encryption key $\mathsf{ek}_i$ in the KeyGen procedure. To do this, party $\mathcal{P}_i$ first generates a keypair and posts the public key to a PKI (line 1-2). It then permutes the $k$-regular graph based on random beacon output and defines its committee as all parties $\mathcal{P}_j$ where the $j$:th vertex is a neighbour to the $i$:th vertex in $G$ (line 3-5). Then, it computes a shared key for each committee member and outputs the PSA encryption key, consisting of the indexes and shared keys for the committee members (line 6-11).

To encrypt a message, $\mathcal{P}_i$ executes the Enc procedure, which outputs the message masked with the value $t_i$. $t_i$ is computed as a sum-of-PRFs for the $i$:th committee. In more detail, for each $\mathcal{P}_j$ in the committee, compute the output of the PRF indexed by the shared key between $\mathcal{P}_i$ and $\mathcal{P}_j$, on input the current label. If $i < j$, the output of the PRF is subtracted from the sum. Otherwise it is added. Thus, each time $\mathcal{P}_i$ adds a random value to its masking value, its neighbour $\mathcal{P}_j$, will subtract *the same value* from its masking value.

The $\mathsf{Aggr}$ procedure computes the sum of all plaintexts by summing all ciphertexts. This will only work if all ciphertexts are included, otherwise, the masking values will not cancel out.

*Correctness:* Let us now prove correctness. By definition:

$$\mathsf{DIPSAUCE.Aggr}\left(\{c_i\}_{i\in n}\right) = \sum_{i\in n} c_i = \sum_{i\in n} m_i + \sum_{i\in n} t_i.$$

Since $G$ is $k$-regular and there exists a one-to-one mapping (bijection) between every vertex $v_i$ and its neighbour set $N(v_i)$, there exist unique indices $i_1,\dots,i_k$ with $i_j \neq i$ for $j = 1\dots,k$, such that $i \in \vec{J}_{i_j}$ for $j = 1,\dots,k$.

Let $i'$ denote any one of the indices $i_j$. Since NIKE is correct – that is, since $\mathsf{NIKE.SharedKey}(\mathsf{pk}_i, \mathsf{sk}_{i'}) = \mathsf{NIKE.SharedKey}(\mathsf{pk}_{i'}, \mathsf{sk}_i)$, we also have:

$$\forall K_i[\ell] : \exists K_{i'}[\ell'] \text{ s.t. } K_i[\ell] = K_{i'}[\ell']$$

Thus $\mathsf{DIPSAUCE}$ is correct if NIKE is correct and $G$ is $k$-regular, since then all $K_i[\ell]$ cancels out during aggregation s.t. $\sum_{i\in n} t_i = 0$.

## 3.3 Security Analysis

We use a similar proof strategy as $\mathsf{LaSS\text{-}PSA}$, originating from Abdalla et al. [1], where we form a hybrid argument from a series of games, where each game changes the definition of the $\mathsf{QLeftRight}$-oracle. Table 1 illustrates the strategy.

The first game, $\mathsf{G}_0$, corresponds to the $\mathsf{AO}_0$-game where $\mathsf{QLeftRight}$ queries are answered with the encryption of $m_i^0$. The last game, $\mathsf{G}_3$, corresponds to the $\mathsf{AO}_1$-game where $\mathsf{QLeftRight}$ queries are answered with the encryption of $m_i^1$. Thus, if the security of the transitions between the games hold, the adversary cannot tell the $\mathsf{AO}_0$-game from the $\mathsf{AO}_1$-game. The transition from $\mathsf{G}_0$ to $\mathsf{G}_1$ consists of adding a *perfect* secret sharing (denoted $\mathsf{PSS}$ in Table 1) of zero to the threshold-sum-of-PRFs, so that all $t_i$ are perfectly random without destroying the correctness of the scheme. This transition is justified if the threshold sum-of-PRFs produces $t_i$ so that it is indistinguishable from randomness. Next, consider the transition from $\mathsf{G}_1$ to $\mathsf{G}_2$, where $c_i$ now encrypts $m_i^1$ instead of $m_i^0$. This transition is justified since $t_i$ is now perfectly random, and thus an adversary cannot distinguish whether $c_i$ is an encryption of $m_i^0$ or $m_i^1$. Finally, the transition from $\mathsf{G}_2$ to $\mathsf{G}_3$ consists of undoing the change made in $\mathsf{G}_1$ (with the same security argument). We arrive at the following theorem.

**Theorem 1.** $\mathsf{DIPSAUCE}$ *is* $\mathsf{AO}$*-secure if* $t_i$ *is indistinguishable from randomness for a computationally bounded adversary except with a negligible advantage.*

**Proving the threshold sum-of-PRFs technique** We now prove that $t_i$ is indistinguishable from randomness to a static malicious adversary.

| Game | Definition of QLeftRight-oracle | Argument |
|---|---|---|
| $\mathsf{G}_0$ | $t_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J}_i[\ell]} \cdot \mathsf{PRF}_{\vec{K}_i[\ell]}(l)$<br>$c_i \leftarrow m_i^0 + t_i$ | |
| $\mathsf{G}_1$ | $\boxed{t'_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J}_i[\ell]} \cdot \mathsf{PRF}_{\vec{K}_i[\ell]}(l)}$<br>$\boxed{t_i \leftarrow t'_i + \mathsf{PSS}(0, i, n - \lvert\mathcal{C}\rvert)}$<br>$c_i \leftarrow m_i^0 + t_i$ | $t_i$ indisting. from rand. |
| $\mathsf{G}_2$ | $t'_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J}_i[\ell]} \cdot \mathsf{PRF}_{\vec{K}_i[\ell]}(l)$<br>$t_i \leftarrow t'_i + \mathsf{PSS}(0, i, n - \lvert\mathcal{C}\rvert)$<br>$c_i \leftarrow \boxed{m_i^1} + t_i$ | one-time-pad info. theo. secure |
| $\mathsf{G}_3$ | $\boxed{t_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J}_i[\ell]} \cdot \mathsf{PRF}_{\vec{K}_i[\ell]}(l)}$<br>$c_i \leftarrow m_i^1 + t_i$ | $t_i$ indisting. from rand. |

Table 1: Strategy for proving AO-Security. A box marks the change in each game.

*Proof Outline:* We first formalize the security of our building blocks NIKE and sum-of-PRFs in the context of our scheme in Lemmas 1 and 2. Intuitively Lemma 1 states that all NIKE derived keys are private to the negotiating parties, and Lemma 2 states that the sum-of-PRF output, $t_i$, is secret to an adversary which corrupts all but one out of the parties in a sum-of-PRFs committee. We then, in Theorem 2, consider the DIPSAUCE method, with $k$-sized committees randomly selected from a population of $n$ parties with a threshold $t$ of corrupt parties. Finally, we conclude with Theorem 3 which formalizes the indistinguishably of $t_i$ as a consequence of the previous theorem and lemmas.

*Proof Details:* First, we restate the security of NIKE in the context of our scheme, *i.e.* that NIKE keys derived for honest committee members do not leak anything to the adversary. As a consequence of the security of NIKE, Lemma 1 is true.

**Lemma 1 (Pseudo-Random Shared Keys).** DIPSAUCE.KeyGen *outputs encryption keys* $\mathsf{ek}_i = (\vec{K}_i, \vec{J}_i)$ *s.t each key* $\vec{K}_i[\ell]$ *is indistinguishable from randomness to a computationally bounded adversary when* $\mathcal{P}_i$ *and the committee counterparty* $\mathcal{P}_{\vec{J}[\ell]}$ *(whose index is defined in* $\vec{J}[\ell]$*) are both honest.*

We also restate the security of the sum-of-PRFs technique in our setting. If a key $\vec{K}_i[\ell]$ is (pseudo)-random (*i.e.* when $\mathcal{P}_{\vec{J}[\ell]}$ is honest), the output of $\mathsf{PRF}_{\vec{K}_i[\ell]}(l)$ is also (pseudo)-random. Then since $t_i$ is the sum of all such values, a single honest $\mathcal{P}_j$ renders $t_i$ (pseudo)-random. Thus, an adversary must corrupt all $k$ parties in the committee to learn anything about $t_i$. We get *Lemma* 2.

**Lemma 2 (Sum-of-PRFs).** *An adversary given* $l$ *and up to* $k-1$ *entries in* $\vec{K}_i$ *has a negligible advantage in distinguishing* $t_i = \sum_{\ell=1}^{k} (-1)^{i<\vec{J}_i[\ell]} \cdot \mathsf{PRF}_{\vec{K}_i[\ell]}(l)$ *from randomness.*

By relying on just Lemma 2, security can only hold against an adversary corrupting up to $t = k - 1$ parties. We therefore transfer from the standard sum-of-PRFs technique to our threshold version. Theorem 2 states that for a random committee, an adversary corrupting up to $t$ parties has a negligible chance to corrupt *all* $k$ committee members of a user with these $t$ corruptions.

In the proof, we first argue that the permutation of the graph is pseudorandom. Then, as a stepping stone, we consider the advantage of an adversary guessing the committee of a *specific* party. Intuitively, if each committee is random, a static adversary's best strategy is to randomly guess the $k$ users in the committee. Finally we put an upper bound on the advantage when attempting to guess the committee of *any* honest user, and fully prove the security of the scheme, by considerering an adversary which attempts to learn *any* $t_i$.

**Theorem 2 (Incorruptible Committee).** DIPSAUCE.KeyGen *outputs* $\mathsf{ek}_i = (\cdot, \vec{J_i})$ *s.t a static adversary allowed to corrupt up to t parties,* $k < t < n$*, has a negligible probability in guessing* $\vec{J'}$ *s.t.* $|\vec{J'}| = k$ *and* $\forall j \in \vec{J'} : j \in \vec{J_i}$*, for* some $i$.

*Proof.* **Graph pseudorandomness**: The permutation $\rho$ is determined by the output $r$ of the randomness beacon. Since $r$ is thus *unbiased* and *unpredictable* to a static $\mathcal{A}$, it cannot predict anything about $\rho$ except with the negligible advantage $Adv_{beacon}$. Then, since $|G| = |\rho|$, $\mathcal{A}$ has a negligible advantage in determining which $\mathcal{P}_i$ is associated with which $v_j \in G$.

**Incorruptability of specific committees**: Consider the number of possible $k$-sized committees and the number of $k$-sized committees an adversary can form from $t$ random corruptions. The number of unordered sets of size $k$ within the $n$ parties is $\binom{n}{k}$. An adversary allowed to corrupt up to $t$ out of $n$ parties can form $\binom{t}{k}$ sets of $k$ corrupt parties. Thus, the probability of obtaining a *specific* $k$-sized committee of a specific party when corrupting $t$ out of $n$ parties is $\frac{\binom{t}{k}}{\binom{n}{k}}$.

**Incorruptability of any committee:** An upper bound on the capability to corrupt *all* members in the committee of *any* honest party for a static adversary allowed to corrupt up to $t$ out of $n$ parties can thus be calculated as $n \cdot \frac{\binom{t}{k}}{\binom{n}{k}}$.

**Synthesis:** In conclusion, the advantage to corrupt all committee members of some party is at most $Adv_{beacon} + n \cdot \frac{\binom{t}{k}}{\binom{n}{k}}$, which is negligible for realistic values of $n, t, k$ (see Appendix A for a discussion on the values of $n, t, k$).

Since a static adversary cannot corrupt all nodes in a committee (Theorem 2), and the sum-of-PRFs technique is secure when at least one committee member is honest (Lemma 2), $t_i$ is indistinguishable from randomness.

**Theorem 3 ($t_i$ Indistinguishability).** *In* DIPSAUCE.Enc*, each* $t_i$ *is indistinguishable from randomness to a static adversary allowed to corrupt up to t parties except with a negligible advantage.*

## 3.4 Security Against a Mobile Adversary

Let us sketch a version of DIPSAUCE which is secure against a *mobile* adversary.

**Modelling mobile security** We model mobile security according to Ostrovsky and Yung [36], allowing corruptions and uncorruptions as follows.

*Epochs:* Time is divided into consecutive *epochs* indexed by a counter.

*Corruptions:* A mobile adversary is allowed to corrupt any party $\mathcal{P}_i$. The adversary must make its selection of corrupt parties *before* an epoch is started, but will gain no information from the corrupt parties *until* that epoch is started. An adversary can additionally *uncorrupt* (leave) a corrupted party. When doing so, the adversary retains all knowledge of secrets learned from that party, but has no further control and learns no further secrets. The total number of corrupt parties at the start of an epoch can never exceed $t$. In this model all parties can be corrupt during some stage of the protocol execution, but the adversary learns secrets from at most $t$ parties during each epoch.

**Mobile security with a PKI** We can trivially achieve mobile security by discarding all secrets and re-executing the Setup and KeyGen procedures at the start of an epoch. Since the Setup and KeyGen procedures are efficient in DIPSAUCE, this modification is feasible in practice. There is a caveat to this though. For brevity we have so far omitted how the PKI trust relation is achieved, *i.e.* how the PKI verifies that a public key actually belongs to the claimed identity. However, when secrets are deleted at the end of an epoch, any secret related to the trust relation with the PKI will also be destroyed. This is necessary to prevent a mobile adversary from using this secret to impersonate previously corrupt parties. How then to maintain a relation with the PKI in between epoch changes?

Ostrovsy and Yung describes two methods of maintaining such trust relations. In the first method, the device is assumed to be able to store a secret key that cannot be learned by an adversary corrupting the device. This can be realized using a Trusted Platform Module (TPM) [43] or trusted execution techniques that provide secure storage [37].

The second method consists of updating keys by generating a new key-pair and posting the new public key signed with the previous secret key. While an adversary can also post a new key signed with the previous key, the system will notice that two such public keys have been published and thus consider the device compromised. This assumes that the adversary cannot suppress messages.

We can thus obtain mobile security for DIPSAUCE as follows. Divide the execution of the protocol into a *setup* phase comprised of the Setup and KeyGen procedures, and an *operational* phase comprised of any number of Enc and Aggr procedures. When an epoch ends, each party erases all secrets except the PKI relation secret and then enters the setup phase once the next epoch begins. In this phase, it awaits the system parameters output from the Setup procedure. It then calls the KeyGen procedure (using one of the PKI relation maintaining methods described above) to generate new secrets. This concludes the setup phase, and initiates the operational phase. We arrive at the following.

**Theorem 4 (Informal).** *Let there be a scheme so that the PKI will not accept more than one $(\mathcal{P}_i, pk_i)$ for each $\mathcal{P}_i$. Further, let there be at least one fresh output from the randomness beacon every epoch. Then the above transformation of* DIPSAUCE *is secure against a mobile adversary, corrupting up to $t$ parties.*

## 4 Experimental Evaluation

In this section we evaluate the performance of DIPSAUCE by implementing it on realistic hardware and measuring its performance. The current state-of-the-art schemes KH-PRF-PSA and LaSS-PSA were only evaluated on Intel i5 CPUs in [44,22], giving little insight into how these schemes perform on realistic hardware. For a fair comparison, we have therefore also implemented these schemes and their suggestions for distributing the setup on the same realistic hardware. The code and raw data from our experiments are available at [12] and [13].

### 4.1 Scenario and Experiments

*Scenario: $n$ Clients* measure a statistic, *e.g.* power, and wants to send the sum of the measurements to a *Server*, without revealing individual measurements.

*Setup:* We have implemented the protocols on CC1352 devices with ARM Cortex M4 processors, utilizinge their hardware acceleration of AES, ECC and SHA256. These devices can be considered “mid-range" constrained devices, as they are classified as C3 devices in [10], and are advertised as being suitable for smart-metering. Further details on CC1325 is given in the full version of this paper [11].

*Experiments:* We evaluate the client side efficiency of LaSS-PSA, KH-PRF-PSA and DIPSAUCE by measuring the execution time of the Enc and Setup + KeyGen procedures. For Enc, time is measured from the start of the procedure until the ciphertext is ready to be transmitted. No network overhead is measured for Enc, since all schemes return ciphertexts as random numbers in $\mathbb{Z}_R$ and thus have equivalent network overhead. For Setup + KeyGen, time is measured from the start of the process, including the time needed to transfer data, such as keys, over the network. In the experiments in [44,22], the number of clients (*i.e.* $n$) tested are groups of 1000 to 10000 clients in even increments of 1000. Our tests are done for $n = 1024$, 2025, 3025, 4096, 5041, 6084, 7056, 8100, 9025 and 10000. These sizes are selected to be comparable with previous work, while remaining compatible with requirements in our specific implementation of the DIPSAUCE protocol, which has additional requirements on the group sizes as explained in Section 4.2. Each experiment was repeated 10 times for each group size. Our results are the average of these runs.

### 4.2 Implementations

**DIPSAUCE** We have implemented the graph $G$ as a rook's graph. As a consequence all $n$ must be square numbers and $k = 2\sqrt{n} - 1$. We remark that this is an implementation property, and that regular graphs for other $k, n$ can be efficiently generated [33]. The KeyGen procedure is straightforwardly implemented according to Protocol 1, using a Python based PKI with a CoAP [38] interface, instantiating Beacon as the Drand service [20], and instantiating NIKE as ECDH on the P-256 curve. The Enc instantiates the PRF using AES-128. Both AES-128 and ECDH P-256 utilizes the hardware acceleration of the CC1352 platform.

**KH-PRF-PSA and LaSS-PSA** We here give details on security parameters, chosen instantiations of primitives, and hardware acceleration. For all details on these schemes and our implementations, see the full version of this paper [11].

KH-PRF-PSA*:* We have implemented the KH-PRF-PSA scheme in [22, Sec. 4] and their proposal for a distributed setup in [22, Sec. 5.1]. The implementation uses security parameters $\lambda = 2096$, $q = 2^{128}$ and $p = 2^{85}$. In [22], KH-PRF-PSA uses a hash based KH-PRF, instantiated as SHA3-512. For a fair comparison, we however select a more efficient hash function, SHA256, which is hardware accelerated on the CC1352 platform.

LaSS-PSA We have implemented the LaSS-PSA scheme in [44, Sec. 4] and the proposal for a distributed setup in [44, Sec. 7]. The implementation uses security parameter $\lambda = 128$. We here implement the version which instantiates the PRF using AES-128, since its the most efficient instantiation in the measurements of [44], and is hardware accelerated on the CC1352 platform.

## 4.3 Results

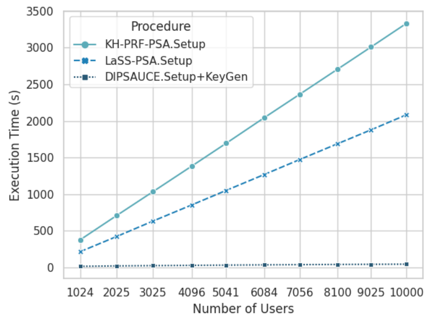

Fig. 2: Setup/KeyGen execution time.

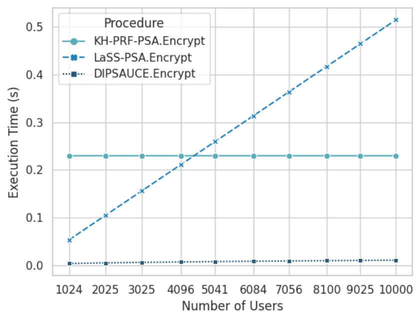

Fig. 3: Encryption execution time.

**Setup and KeyGen** Our evaluation shows that DIPSAUCE significantly outperforms the distributed setups proposed in KH-PRF-PSA and LaSS-PSA in terms of execution time for the setup (and keygen) procedure. We show the execution times in Figure 2. The slope of the graph indicates that DIPSAUCE will have the shortest execution time for all number of users in the system. The execution time of DIPSAUCE grows with the number of users at rate of 3.2 ms per user, a lower rate than KH-PRF-PSA which grows with 330 ms per user and LaSS-PSA which grows with 210 ms per user. This is due to DIPSAUCE only generating $k = 2\sqrt{n-1}$ NIKE shared secrets for $n$ users, rather than $n$ derived secrets as in LaSS-PSA, and LaSS-PSA in turn, being more efficient than KH-PRF-PSA. Compared to LaSS-PSA, DIPSAUCE shows a speedup of 66x.

**Encrypt** Our evaluation of the Enc procedures shows that DIPSAUCE outperform KH-PRF-PSA and LaSS-PSA for all measured number of users in the system. We show the measured execution times of the encrypt procedure in Figure 3. LaSS-PSA and DIPSAUCE have execution times linear in the number of users. The execution time of the Enc procedure grows with 0.052 ms per user for LaSS-PSA and with 0.00075 ms per user for DIPSAUCE. The speedup per user of DIPSAUCE compared to LaSS-PSA is 69x.

KH-PRF-PSA shows a constant execution time of 230 ms for any number of users in the system. Thus, it will outperform DIPSAUCE for large numbers of users. Extrapolating from the measured times, this occurs when $n \approx 300000$.

## 5 Conclusion

In this paper we have showed state-of-the-art PSA schemes and their proposals for a ditributed setup, and found them practically infeasible due to computational complexity which grows with the number of users. To address this, we have provided a formal definition of PSA with a distributed setup, suggested a new PSA scheme adhering to this definition, proved it secure and implemented it on realistic hardware. We found its performance sufficient to be deployed in practice. Let us further elaborate on the following discussion point.

**Client Failures** In a secure PSA scheme, nothing is learned by the aggregator unless all ciphertexts are included in an aggregation. Therefore, a dropped message from an honest client will prevent the aggregator from learning anything. We note that there is a general *non-interactive* mitigation to this practical problem [15] for dealing with client errors, which is applicable to *all* PSA schemes including ours. This however increases computational and network costs. Since the setup in DIPSAUCE is efficient, another alternative to deal with client failures can be to exclude failing clients from the protocol and re-execute the setup, if the failures are fairly infrequent.

## Acknowledgements

We would like to thank Paul Stankovski Wagner, Elena Pagnin and Christian Gehrmann for valuable discussions of this work.

## References

1. Abdalla, M., Benhamouda, F., Gay, R.: From single-input to multi-client inner-product functional encryption. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 552–582. Springer, Berlin, Germany (2019)
2. Abdalla, M., Benhamouda, F., Kohlweiss, M., Waldner, H.: Decentralizing inner-product functional encryption. In: IACR International Workshop on Public Key Cryptography. pp. 128–157. Springer, Berlin, Germany (2019)
3. Becker, D., Guajardo, J., Zimmermann, K.H.: Revisiting private stream aggregation: Lattice-based psa. In: NDSS. Internet Society, Reston, VA, USA (2018)
4. Bell, J.H., Bonawitz, K.A., Gascón, A., Lepoint, T., Raykova, M.: Secure single-server aggregation with (poly) logarithmic overhead. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1253–1269 (2020)
5. Benhamouda, F., Joye, M., Libert, B.: A new framework for privacy-preserving aggregation of time-series data. ACM Transactions on Information and System Security (TISSEC) **18**(3), 1–21 (2016)
6. Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečnỳ, J., Mazzocchi, S., McMahan, B., et al.: Towards federated learning at scale: System design. Proceedings of machine learning and systems **1**, 374–388 (2019)
7. Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H.B., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for privacy-preserving machine learning. In: proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1175–1191 (2017)
8. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Theory of Cryptography Conference. pp. 253–273. Springer, Berlin, Germany (2011)
9. Bormann, C., Ersue, M., Keranen, A.: Terminology for constrained-node networks. RFC 7228, RFC Editor (May 2014), `http://www.rfc-editor.org/rfc/rfc7228.txt`, `http://www.rfc-editor.org/rfc/rfc7228.txt`
10. Bormann, C., Ersue, M., Keränen, A., Gomez, C.: Terminology for Constrained-Node Networks. Internet-Draft draft-ietf-lwig-7228bis-00, Internet Engineering Task Force (Jun 2022), `https://datatracker.ietf.org/doc/draft-ietf-lwig-7228bis/00/`, work in Progress
11. Brorsson, J., Gunnarsson, M.: Dipsauce: Efficient private stream aggregation without trusted parties. Cryptology ePrint Archive, Paper 2023/214 (2023), `https://eprint.iacr.org/2023/214`
12. Brorsson, J., Gunnarsson, M.: Protocol and experiments (2023), `https://github.com/Gunzter/DIPSAUCE-contiki-ng`
13. Brorsson, J., Gunnarsson, M.: Results and corresponding raw data (2023), `https://github.com/Gunzter/practical_psa_results`
14. Castellucci, R.: libtprpg (2013), `https://github.com/ryancdotorg/libtprpg`
15. Chan, T.H.H., Shi, E., Song, D.: Privacy-preserving stream aggregation with fault tolerance. In: International Conference on Financial Cryptography and Data Security. pp. 200–214. Springer, Berlin, Germany (2012)
16. Chase, M., Chow, S.S.: Improving privacy and security in multi-authority attribute-based encryption. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 121–130. ACM, New York, NY, USA (2009)

17. Choi, K., Manoj, A., Bonneau, J.: Sok: Distributed randomness beacons. Cryptology ePrint Archive, Paper 2023/728 (2023), `https://eprint.iacr.org/2023/728`, `https://eprint.iacr.org/2023/728`
18. Chotard, J., Dufour Sans, E., Gay, R., Phan, D.H., Pointcheval, D.: Decentralized multi-client functional encryption for inner product. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 703–732. Springer, Berlin, Germany (2018)
19. Chotard, J., Dufour-Sans, E., Gay, R., Phan, D.H., Pointcheval, D.: Dynamic decentralized functional encryption. In: Annual International Cryptology Conference. pp. 747–775. Springer, Berlin, Germany (2020)
20. Drand: Drand - a distributed randomness beacon daemon (2022), `https://github.com/drand/drand`
21. Emura, K.: Privacy-preserving aggregation of time-series data with public verifiability from simple assumptions. In: Australasian Conference on Information Security and Privacy. pp. 193–213. Springer, Berlin, Germany (2017)
22. Ernst, J., Koch, A.: Private stream aggregation with labels in the standard model. Proc. Priv. Enhancing Technol. **2021**(4), 117–138 (2021)
23. Gope, P., Sikdar, B.: Lightweight and privacy-friendly spatial data aggregation for secure power supply and demand management in smart grids. IEEE Transactions on Information Forensics and Security **14**(6), 1554–1566 (2018)
24. IEEE Computer Society: IEEE Standard for Local and Metropolitan Area Networks, Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) (10 2011)
25. Instruments, T.: Launchxl-cc1352r1 simplelink™ multi-band cc1352r wireless mcu launchpad™ development kit. `https://www.ti.com/tool/LAUNCHXL-CC1352R1` (2021), accessed: 2022-01-21
26. Joye, M., Libert, B.: A scalable scheme for privacy-preserving aggregation of time-series data. In: International Conference on Financial Cryptography and Data Security. pp. 111–125. Springer, Berlin, Germany (2013)
27. Kabalci, Y.: A survey on smart metering and smart grid communication. Renewable and Sustainable Energy Reviews **57**, 302–318 (2016)
28. Kursawe, K., Danezis, G., Kohlweiss, M.: Privacy-friendly aggregation for the smart-grid. In: International Symposium on Privacy Enhancing Technologies Symposium. pp. 175–191. Springer, Berlin, Germany (2011)
29. Laurie, B.: Certificate transparency. Communications of the ACM **57**(10), 40–46 (2014)
30. Leontiadis, I., Elkhiyaoui, K., Molva, R.: Private and dynamic time-series data aggregation with trust relaxation. In: International Conference on Cryptology and Network Security. pp. 305–320. Springer, Berlin, Germany (2014)
31. Lyu, L., Nandakumar, K., Rubinstein, B., Jin, J., Bedo, J., Palaniswami, M.: Ppfa: Privacy preserving fog-enabled aggregation in smart grid. IEEE Transactions on Industrial Informatics **14**(8), 3733–3744 (2018)
32. Ma, Y., Woods, J., Angel, S., Polychroniadou, A., Rabin, T.: Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 477–496. IEEE Computer Society (2023)
33. Meringer, M.: Fast generation of regular graphs and construction of cages. Journal of Graph Theory **30**(2), 137–146 (1999)
34. Molina-Markham, A., Shenoy, P., Fu, K., Cecchet, E., Irwin, D.: Private memoirs of a smart meter. In: Proceedings of the 2nd ACM workshop on embedded sensing

systems for energy-efficiency in building. pp. 61–66. ACM, New York, NY, USA (2010)
35. Oikonomou, G., Duquennoy, S., Elsts, A., Eriksson, J., Tanaka, Y., Tsiftes, N.: The Contiki-NG open source operating system for next generation IoT devices. SoftwareX **18**, 101089 (2022). https://doi.org/https://doi.org/10.1016/j.softx.2022.101089
36. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks. In: Proceedings of the tenth annual ACM symposium on Principles of distributed computing. pp. 51–59. ACM, New York, NY, USA (1991)
37. Pinto, S., Santos, N.: Demystifying arm trustzone: A comprehensive survey. ACM computing surveys (CSUR) **51**(6), 1–36 (2019)
38. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (Jun 2014). https://doi.org/10.17487/RFC7252, https://www.rfc-editor.org/info/rfc7252
39. Shi, E., Hubert Chan, T.H., Rieffel, E., Chow, R., Song, D.: Privacy-preserving aggregation of time-series data. Network and Distributed System Security Symposium, NDSS **1**, 17 (2011)
40. Szigeti, J.: Big unsigned integers (2022), https://github.com/SzigetiJ/biguint
41. Takeshita, J., Carmichael, Z., Karl, R., Jung, T.: Terse: Tiny encryptions and really speedy execution for post-quantum private stream aggregation. In: Li, F., Liang, K., Lin, Z., Katsikas, S.K. (eds.) Security and Privacy in Communication Networks. pp. 331–352. Springer Nature Switzerland, Cham (2023)
42. Takeshita, J., Karl, R., Gong, T., Jung, T.: Slap: Simple lattice-based private stream aggregation protocol. Cryptology ePrint Archive, Paper 2020/1611 (2020), https://eprint.iacr.org/2020/1611, https://eprint.iacr.org/2020/1611
43. TCG: TCG TPM Specification Version 1.2 - Part 1 Design Principles. Tech. rep., TCG, Beaverton, OR, United States (March 2011)
44. Waldner, H., Marc, T., Stopar, M., Abdalla, M.: Private stream aggregation from labeled secret sharing schemes. Cryptology ePrint Archive, Paper 2021/081 (2021), https://eprint.iacr.org/2021/081, https://eprint.iacr.org/2021/081

## Appendix

## A Adversary Advantage

The adversary advantage (excluding the potential advantage resulting from the beacon) is calculated as $n \cdot \frac{\binom{t}{k}}{\binom{n}{k}}$ in Theorem 2. Table 2 shows this advantage for realistic $n$, $t$ and $k$, where $t = n/2$ and $k = 2\sqrt{n} - 2)$ in a rook's graph which is the $k$-regular graph which was used in our implementation.

Code to calculate this advantage for different values is available below.

```
#rook's graph adversary advantage
import math
from decimal import Decimal

for n in [1024, 2025, 3025, 4096, 5041, 6084, 7056, 8100, 9025, 10000]:
```

| $n$ | $k$ | $t$ | Advantage |
|---|---|---|---|
| 1024 | 62 | 512 | $2^{-55}$ |
| 2025 | 88 | 1012 | $2^{-78}$ |
| 3025 | 108 | 1512 | $2^{-99}$ |
| 4096 | 126 | 2048 | $2^{-117}$ |
| 5041 | 140 | 2520 | $2^{-131}$ |
| 6084 | 154 | 3042 | $2^{-144}$ |
| 7056 | 166 | 3528 | $2^{-156}$ |
| 8100 | 178 | 4050 | $2^{-168}$ |
| 9025 | 188 | 4512 | $2^{-178}$ |
| 10000 | 198 | 5000 | $2^{-188}$ |

Table 2: Adversary advantage in DIPSAUCE with a rook's graph given by $n \cdot \frac{\binom{t}{k}}{\binom{n}{k}}$ for different values of $n$ and a corruption ratio of 0.5.

```
# number of columns/rows
x=float(math.sqrt(n))
#corruption threshold
thresh = 0.5

k = 2*x-2
t = n*thresh

nc = Decimal(math.comb(int(t),int(k)))
npc = Decimal(math.comb(int(n),int(k)))

print("all: n =", int(n), ", k =", int(k), ", t =", int(t), ",
    advantage =", Decimal(n) * nc/npc)
```

## B Experimental Setup

### B.1 CC1352R SimpleLink

The CC1352R SimpleLink [25] is a series of micro controllers (MCU) sold by Texas Instruments. Its intended application areas include: building automation, grid infrastructure, water meters, *electricity meters*, gas meters, and personal electronics. It features a 48 MHz ARM Cortex-M4F CPU, with 88KB of RAM and 602KB of ROM. It also features a wide variety of peripherals. Of special interest in this work are the hardware accelerated cryptography peripherals for AES-128, SHA256, Elliptic Curve Diffie-Hellman, and a TRNG. Elliptic Curves on Short Weierstrass form are fully supported and include NIST-P224, NIST-P256, NIST-P384, and NIST-P512, Brainpool-256R1, Brainpool-384RR1, and Brainpool-512R1. Elliptic curves on Montgomery form such as Curve25519 have limited hardware support. The built in TRNG has a self test required by FIPS

140. CC1352R also has the capabilities to run the wireless protocol IEEE802.15.4 Low Power Personal Area Network (LoWPAN).

### B.2 Operating System and Software

The experiments are implemented on the Contiki-NG operating system [35], designed for constrained devices, with a its built in network stack. All hardware accelerated cryptographic operations were performed using the default drivers included in Contiki-NG. Furthermore we used the BigUint128 library [40] to perform 128-bit arithmetics, and the libtprpg [14] library to generate the pseudo-random permutation used in DIPSAUCE.

### B.3 Communication

In our experiments we have used the IEEE 802.15.4 [24] physical layer operating on the 2.4 GHz band. The network stack is the Contiki-NG networking stack with IPv6, UDP and CoAP with default settings. An RPL-border-router is required, since IEEE 802.15.4 is not supported on the laptop we used in the experiments. The RPL-border-router was run on another CC1352R device with the standard RPL-border-router application provided in Contiki-NG.

### B.4 Experimental Setup

We executed the protocols on a CC1352R device, which we denote as the *Client*. The *Client* communicates with a *Server* running on a laptop. The RPL-border-router is connected to the laptop with a USB cable. The *Client* can then communicate with the *Server* running on the laptop via the border-router. The *Client* and the RPL-border-router were placed close to each other, with the antennas facing each other to minimize packet-loss. Figure 4 illustrates the setup.

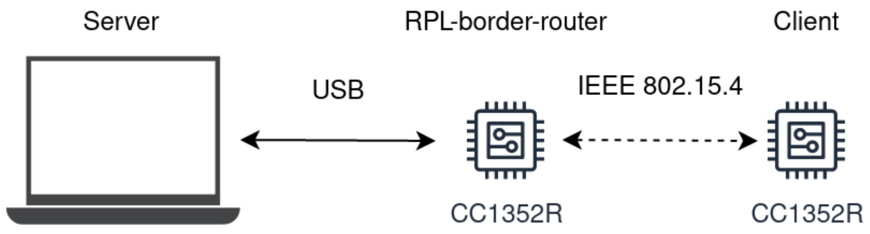

Fig. 4: An illustration of the experimental setup.

## C Full Definitions of Existing Schemes

**Private Stream Aggregation:** We here recall the formal definition of PSA.

**Definition 4 (Private Stream Aggregation).** *Let $R \in \mathbb{N}, \mathcal{M} = \mathbb{Z}_R$ and $\mathcal{L}$ be a set of labels. A PSA scheme is defined by the procedures:*

- $\mathsf{Setup}(\lambda, n)$*: On input the security parameter $\lambda$ and the number of parties $n$, output public parameters $\mathsf{pp}$ and $n+1$ secret symmetric encryption keys $\{\mathsf{ek}_i\}_{i \in [n+1]}$ (where $\mathsf{ek}_{n+1}$ is the aggregator key, sometimes alternatively denoted as $\mathsf{ek}_a$).*
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, m_i, l)$*: On input the public parameters $\mathsf{pp}$, an encryption key $\mathsf{ek}_i, i \leq n$, a message $m_i \in \mathcal{M}$ and a label $l \in \mathcal{L}$, output the ciphertext $c_i$.*
- $\mathsf{Aggr}(\mathsf{pp}, \mathsf{ek}_a, \{c_i\}_{c \in [n]}, l)$*: Given the public parameters $\mathsf{pp}$, the aggregator key $\mathsf{ek}_a$, a set of $n$ ciphertexts $\{c_i\}_{c \in [n]}$, and a label $l$, it outputs the sum of all plaintexts, $M \pmod{R}$.*

A $\mathsf{PSA}$ scheme must satisfy *correctness*. Correctness is satisfied if, for any $n, \lambda \in \mathbb{N}, m_1, \ldots, m_n \in \mathbb{Z}_R$ and any label $l \in \mathcal{L}$, so that $(\mathsf{pp}, \{\mathsf{ek}_i\}_{i \in [n+1]}) \leftarrow \mathsf{Setup}(\lambda, n)$, and $\forall \{c_i\}_{i \in [n]} : c_i = \mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, l, m_i)$:

$$\mathsf{AggrDec}(\mathsf{pp}, \mathsf{ek}_a, l, \{c_i\}_{i \in [n]}) = \sum_{i \in [n]} m_i \pmod{R}$$

Further, a *secure* $\mathsf{PSA}$ scheme must satisfy Aggregator Obliviousness (AO). The below definition of AO regards *encrypt-once* security, where a client only encrypt one value per label.

**Definition 5 (Aggregator Obliviousness).** *Let $\mathsf{PSA}$ be a* PSA *scheme. Let the experiment $AO_b$ in Figure 5 be defined with the following oracles:*

- $\mathsf{QCorrupt}(i)$*: The oracle outputs the encryption key $\mathsf{ek}_i$ of user $i$. For $i = n+1$, it outputs the aggregator key $\mathsf{ek}_a$.*
- $\mathsf{QEnc}(i, m_i, l^*)$*: The oracle outputs $ct_i = \mathsf{Enc}(\mathsf{ek}_i, m_i, l^*)$ on a query.*
- $\mathsf{QChallenge}(\mathcal{U}, \{m_i^0\}_{i \in \mathcal{U}}, \{m_i^1\}_{i \in \mathcal{U}}, l^*)$*: The adversary specifies a set of user indices $\mathcal{U} \subseteq [n]$, a label $l^*$ and two challenge messages for each user from $\mathcal{U}$. The oracle answers with encryptions of $m_i^b$, that is $\{c_i \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, m_i^b, l^*)\}_{i \in \mathcal{U}}$. This oracle can only be queried once during the game. If the adversary does not query this oracle, $\mathcal{U} = \emptyset$.*

*At the end of the game, the adversary $\mathcal{A}$ outputs a guess $\alpha$, whether $b = 0$ or $b = 1$. A PSA scheme is* Aggregator Oblivious*, if an adversary $\mathcal{A}$ for all sufficiently large $\lambda$, has a negligible advantage in winning the game.*

The adversary advantage is defined as:

$$\mathsf{Adv}_{\mathcal{A},\mathsf{PSA}}^{AO}(\lambda, n) =\mid Pr[\mathsf{AO}_O(\lambda, n, \mathcal{A}) = 1] - Pr[\mathsf{AO}_1(\lambda, n, \mathcal{A}) = 1] \mid$$

```
$\mathsf{AO}_b(\lambda, n, \mathcal{A})$

$(\mathsf{pp}, \{\mathsf{ek}_i\}_{i\in[n+1]}) \leftarrow \mathsf{Setup}(\lambda, n)$
$\alpha \leftarrow A^{\mathsf{QCorrupt}(\cdot),\mathsf{QEnc}(\cdot),\mathsf{QChallenge}(\cdot,\cdot,\cdot,\cdot)}(\mathsf{pp})$
if condition $(*)$ is satisfied then
    Output $\alpha \overset{?}{=} b$
else
    Output 0
end if
```

Fig. 5: The aggregator obliviousness game defining security for a PSA scheme.

Consider the following sets:

– Let $\mathcal{E}_l^* \subseteq [n]$ be the set of all users for which $\mathcal{A}$ has asked an encryption query on label $l$.
– Let $\mathcal{CS} \subseteq [n]$ be the set of *users* for which $\mathcal{A}$ has asked a corruption query. Even if the aggregator is corrupted, this set only contains the corrupted *users* and not the aggregator.
– Let $\mathcal{Q}_{l^*} := \mathcal{U} \cup \mathcal{E}_{l^*}$ be the set of users for which $\mathcal{A}$ asked a challenge or encryption query on label $l^*$.

*Condition* $(*)$ is *satisfied* (as used in Figure 5), if all of the following conditions are satisfied:

– $\mathcal{U} \cap \mathcal{CS} = \emptyset$. This means that all users for which $\mathcal{A}$ received a challenge ciphertext must stay uncorrupted during the entire game.
– $\mathcal{A}$ has not queried $\mathsf{QEnc}(i, m_i, l^*)$ twice for the same $(i, l^*)$.
– $\mathcal{U} \cap \mathcal{E}_{l^*} = \emptyset$. This means that $\mathcal{A}$ is allowed to get a challenge ciphertext only from users for which they ask an encryption query on the challenge label $l^*$. Doing so would violate the encrypt-once restriction.
– If $\mathcal{A}$ has corrupted the aggregator *and* $\mathcal{Q}_{l^*} \cup \mathcal{CS} = [n]$ the following equality must hold in order to prevent trivial wins by using the knowledge of the aggregators knowledge of the sum of all honest parties plaintexts.

$$\sum_{i\in U} x_i^0 = \sum_{i\in U} x_i^1$$

This condition is called the *balance-condition.*

## D Details on the Evaluation of the performance of state-of-the-art PSA schemes

### D.1 Protocol Definitions and Implementations

**KH-PRF-PSA** We recall the definition of the KH-PRF-PSA protocol from [22] in Protocol 2.

**Protocol 2** – The KH-PRF-PSA scheme in [22].

```
Setup($\lambda, n$):
1: $\forall i \in [n] : \mathsf{ek}_i \xleftarrow{\$} \mathbb{Z}_R^\lambda$
2: $\mathsf{ek}_a = \sum_{i\in[n]} \mathsf{ek}_i$
3: return $\mathsf{ek}_a, \{\mathsf{ek}_i\}_{i\in[n]}$

Enc($\mathsf{ek}_i, m_i, l$):
1: $t_i = \mathsf{KH\text{-}PRF}_{\mathsf{ek}_i}(l)$
2: $c_i = (t_i + m_i) \pmod R$
3: return $c_i$

Aggr($\mathsf{ek}_a, \{c_i\}_{i\in[n]}, l$):
1: $m_a = \sum_{i\in n} c_i - \mathsf{KH\text{-}PRF}_{\mathsf{ek}_a}(l) \pmod R$
2: return $m_a$
```

We have implemented the KH-PRF-PSA realization in [22, Sec. 4], which uses a hash based KH-PRF secure in the Random Oracle Model. The implementation uses parameter $R \in \mathbb{N}$ and security parameters $\lambda = 2096$, $q = 2^{128}$ and $p = 2^{85}$. The encryption key ek is a vector of $\lambda$ elements in $\mathbb{Z}_q$. Its size is thus $\lambda \cdot q = 33536$ bytes.

The KH-PRF is defined as the inner product of the ek and the output of the function $H'(l)$ (where $l$ is the given label):

$$\mathsf{KH\text{-}PRF}_{\mathsf{ek}}(l) = \lfloor \langle H'(l), \mathsf{ek} \rangle \rfloor_p$$

This definition uses the syntax of [22], where: $\lfloor x \rceil_p = \lfloor x \cdot p/q \rfloor$. The function $H'(l)$ is defined as a vector of $\lambda$ hashes of the label concatenated with a counter, and reduced modulo $q$:

$$H'(l) = \begin{pmatrix} H(l||\text{""}||\text{"}1\text{"}) \pmod q \\ \vdots \\ H(l||\text{""}||\text{"}\lambda\text{"}) \pmod q \end{pmatrix}$$

In the instantiation in [22] $H$ is instantiated as SHA3-512. For a fair comparison, we however select a more efficient hash function, SHA256, which is hardware accelerated on the CC1352 platform.

**LaSS-PSA** Let us also recall the LaSS-PSA scheme from [44], presented in Protocol 3. The LaSS-PSA realization is presented using the notation $(-1)^{(i<j)}$, introduced in Section 2. The realization uses parameter $R \in \mathbb{N}$ and the security parameter $\lambda = 128$. Note that $K_{i,i}$ is left undefined.

LaSS-PSA uses LaSS to mask the message. We here implement the version which instantiates the PRF using AES-128, since its the most efficient instantiation of LaSS in the measurements of [44], and is hardware accelerated on the

**Protocol 3** – The LaSS-PSA scheme [44].

```
Setup($\lambda, n$):
1: $\forall i \in [n+1], \forall j$ s.t. $n \leq j > i : K_{i,j} \xleftarrow{\$} \mathbb{Z}_R$
2: $\forall i \in [n+1], \forall j$ s.t. $n \leq j < i : K_{i,j} = K_{j,i}$
3: let $\mathsf{ek}_i = \vec{K}_i$ be the vector s.t. $\forall j \in [n] : \vec{K}_i[j] = K_{i,j}$
4: return $\{\mathsf{ek}_i\}_{i \in [n+1]}$

Enc($\mathsf{ek}_i = \vec{K}_i, m_i, l$):
1: $t_i \leftarrow \sum_{j \in [n+1] \setminus \{i\}} (-1)^{i<j} \cdot \mathsf{PRF}_{\vec{K}_i[j]}(l)$
2: $c_i = (t_i + m_i) \pmod R$
3: return $c_i$

Aggr($\mathsf{ek}_a = \vec{K}_{n+1}, \{c_i\}_{i \in [n]}$):
1: $m_a = \sum_{i \in n} c_i + \sum_{j \in [n]} \mathsf{PRF}_{\vec{K}_{n+1}[j]}(l) \pmod R$
2: return $m_a$
```

CC1352 platform. The encryption key ek consists of $n$ elements from $\mathbf{Z}_R$, where $n$ is the number of users, and thus has size $n \cdot log_2(R)$.

### D.2 Results

The results of the experiments are available in Figure 6. The KH-PRF construction used in KH-PRF-PSA has an execution time independent of the number of parties. Our measurements show this constant execution time to be 230 ms. The execution time of LaSS-PSA is linear with a coefficient of 0.05 ms per party in the system. The lines intersect at approximately 4200 users. These trends correspond to the results from [22, Section 4.4], but the execution times for both KH-PRF-PSA and LaSS-PSA are around 200x longer in our measurements compared to the numbers presented in [22]. This is since our experiments are executed on a constrained devices whereas the experiments in [22] are executed on an Intel Core i5 CPU.

## E Details on the Evaluation of the Existing Proposed Methods for a Distributed Setup

Recall that all previous PSA schemes, including KH-PRF-PSA and LaSS-PSA, are presented with a trusted party for key distribution. Both KH-PRF-PSA [22] and LaSS-PSA [44], briefly discuss an approach to distribute the setup by negotiating a key between each party in the scheme, but does not give details on how to do this. We here give details on how a distributed setup procedure for KH-PRF-PSA and LaSS-PSA can be constructed using the proposed method, implement the resulting schemes on the CC1352 platform, and evaluate the performance of the client side operations (*i.e.* both setup and encryption since the setup is now

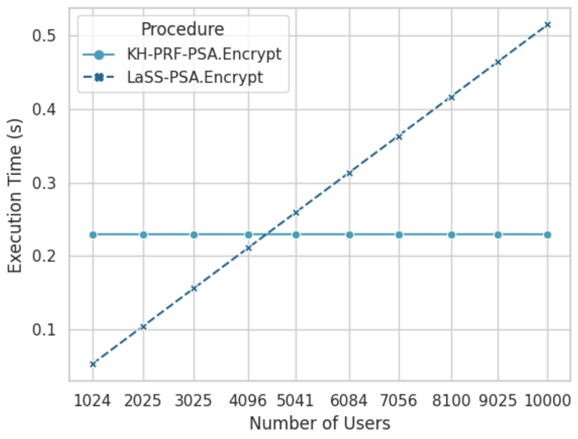

Fig. 6: Execution time in seconds of the Enc procedure in KH-PRF-PSA and LaSS-PSA.

performed by the clients instead of by a trusted party). This gives us an estimate for the performance of this approach. The code for our experiments and protocol implementation, and the raw data of the results is available at [12]. The raw data of the results are available at [13].

It is not our intent to prove the security of these distributed setups. We only wish to show their (in)efficiency.

### E.1 A Distributed Setup for KH-PRF-PSA

Ernst and Koch propose a decentralized setup protocol in [22, Section 5.1] based on the sum-of-PRF technique.

**Decentralizing the Protocol:** Protocol 4 introduces a detailed realization of the suggested approach to decentralize the KH-PRF-PSA protocol. Note that in order for this distributed setup to be compatible with Protocol 2, the evaluator must derive $k_a$ as the sum of all $\mathsf{aks}_i$, *i.e.* $k_a = \sum_{i \in [n]} \mathsf{aks}_i$.

In our implementation of Protocol 4, we use a python based PKI with a CoAP [38] interface where all keys of other parties are registered. We have instantiated NIKE using ECDH P-256, which is hardware accelerated on the CC1352 platform. The PRF was instantiated using hardware accelerated AES-128.

### E.2 A Distributed Setup for LaSS-PSA

Waldner et al. propose a decentralized setup protocol in [44, Sec. 7], which we give details on how to construct in Protocol 5. Note that to make this setup compatible with Protocol 3, the aggregator must also execute the setup protocol.

Our implementation of Protocol 5 uses a python based PKI with a CoAP [38] interface where all public keys are registered. We have instantiated NIKE using

**Protocol 4** – Distributed Setup for KH-PRF-PSA

```
Setup($\lambda, n, i$):

 1: let $\vec{E}_i$ be a vector where $\forall j \in \{1, \ldots, \lambda\} : \vec{E}_i[j] \xleftarrow{\$} \mathbb{Z}_R$
 2: $\mathsf{ek}_i \leftarrow \vec{E}_i[1]||\ldots||\vec{E}_i[\lambda]$
 3: $\mathsf{npp} \leftarrow \mathsf{NIKE.Setup}(\lambda)$
 4: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{NIKE.KeyGen}(\mathsf{npp})$
 5: Post $(\mathcal{P}_i, \mathsf{pk}_i)$ to the PKI
 6: Wait until the PKI returns a $\mathsf{pk}_j$ for each $j \in [n]$
 7: for $j \in [n] \setminus \{i\}$ do
 8:     $K_j \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_j, \mathsf{sk}_i)$
 9: end for
10: for $\ell \in \{1, \ldots, \lambda\}$ do
11:     $b_{i,\ell} \leftarrow \sum_{j \in [n] \setminus \{i\}} (-1)^{i<j} \cdot \mathsf{PRF}_{K_j}(\ell)$
12:     $\vec{A}_i[\ell] = \vec{E}_i[\ell] + b_{i,\ell} \pmod R$
13: end for
14: $aks_i \leftarrow \vec{A}_i[1]||\ldots||\vec{A}_i[\lambda]$
15: return $\mathsf{ek}_i, \mathsf{aks}_i$
```

**Protocol 5** – Distributed Setup for LaSS-PSA.

```
Setup($\lambda, n, i$):

1: $\mathsf{npp} \leftarrow \mathsf{NIKE.Setup}(\lambda)$
2: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{NIKE.KeyGen}(\mathsf{npp})$
3: Post $(\mathcal{P}_i, \mathsf{pk}_i)$ to the PKI
4: Wait until the PKI returns a $\mathsf{pk}_j$ for each $\mathcal{P}_j \in \mathcal{P}$
5: for $j \in [n] \setminus \{i\}$ do
6:     $K_j \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_j, \mathsf{sk}_i)$
7: end for
8: return $\mathsf{ek}_i = \vec{K}_i$
```

ECDH P-256. ECDH P-256 is hardware accelerated on the CC1352 platform. The PRF was instantiated using hardware accelerated AES-128.

## E.3 Experiments and Results

The setup and experiments described here are the same as those described in Section 4.1 and Appendix B, however instead of measuring Enc, we measure the Setup execution times. The results are available in Figure 7. The figure shows that the execution times of the parts of KH-PRF-PSA and LaSS-PSA grow linearly with the number of users in the system. The coefficient for KH-PRF-PSA is higher than that for LaSS-PSA. The reason for this difference in performance is that KH-PRF-PSA, in addition to deriving pairwise shared keys for all users (which is done in both Protocol 4 and Protocol 5), also generates a larger secret key $ek_i$ (steps 1-2 in Protocol 4) and masks $ek_i$ before it is sent to the aggregator (steps 10-14 in Protocol 4). Thus LaSS-PSA outperforms KH-PRF-PSA for all number of users in the system.

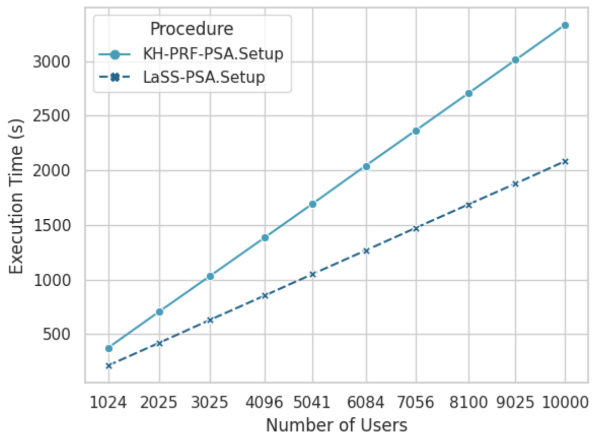

Fig. 7: Execution time in seconds of the Setup procedure in KH-PRF-PSA and LaSS-PSA.