DIPSAUCE: Efficient Private Stream Aggregation Without Trusted Parties

Joakim Brorsson
Lund University
Lund, Sweden
joakim.brorsson@eit.lth.se

Martin Gunnarsson
RISE, Research Institutes of Sweden AB
Lund, Sweden
martin.gunnarsson@ri.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. Due to this ability to efficiently enable central data analytics without leaking individual user data, PSA schemes are often used for IoT data analytics scenarios where privacy is important, such as smart metering. However, all known PSA schemes require a trusted party for key generation, which is undesirable from a privacy standpoint. Further, even though the main benefit of PSA schemes over alternative technologies such as Functional Encryption is that they are efficient enough to run on IoT devices, there exists no evaluation of the efficiency of existing PSA schemes on realistic IoT devices.

In this paper, we address both these issues. We first evaluate the efficiency of the state of the art PSA schemes on realistic IoT devices. We then propose, implement and evaluate a DIstributed setup PSA scheme for Use in Constrained Environments (DIPSAUCE). DIPSAUCE is the first PSA scheme that does not rely on a trusted party. Our security and efficiency evaluation shows that it is indeed possible to construct an efficient PSA scheme without a trusted central party. Surprisingly, our results also show that, a 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, PSA, Data Analytics, IoT, Smart Metering, sum-of-PRFs

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 can calculate 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 [35]. Thus the central party is often considered untrusted and cannot be given individual data readings. There exist works studying how to centrally derive statistics without revealing individual data points for specific use cases (*e.g.* [22, 28, 31] in the case of smart meters). We are however interested in developing general techniques for IoT data analytics.

IoT devices are often *constrained* [7], *i.e.* they have one or more of the following characteristics: low computational power and memory, operate over low throughput lossy networks or are battery powered. The computational and network cost of any scheme for constrained devices is therefore of high importance, and the performance cost of privacy enhancing technologies needs to be minimal for IoT protocols.

Functional Encryption. A first technique that comes to mind for IoT data analytics is that of Functional Encryption (FE) [5], which allows for evaluating a function on encrypted data if the evaluating party knows a *functional decryption key* for that function. 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, and allows a central party to evaluate a function on the encrypted data. However, even the most efficient DMCFE schemes [2, 17, 18], 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.

Private Steam Aggregation. In use cases which only require evaluating the sum of encrypted inputs (*e.g.* smart metering) rather than a general function or the inner product, we can look to PSA for more efficient constructions. The notion of PSA was introduced in [41] and is a similar concept to (D)MCFE. However it is restricted to computing sums rather than inner products (or general functions), which allows more efficient constructions.

Both in the original PSA scheme [41] and in follow up works [3, 4, 14, 20, 21, 26, 30, 43–45], the setup (which includes key generation) relies upon a *trusted party*. Such a design choice erodes trust in a privacy enhancing technology and is particularly engraving in the case of PSA schemes, since the purpose of PSA is to avoid a central party with access to individual data.

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. To the best of the authors knowledge, none of the known PSA schemes avoids a trusted party. Notably, in current state of the art PSA schemes [21, 45], there are brief discussions on how to modify the schemes to avoid a trusted party by using a distributed setup inspired by the DMCFE scheme in [18]. However, neither work has any formal protocol description, security evaluation or efficiency evaluation of the proposed modification. In Section 3 we show that such modifications are too inefficient for constrained environments. There is thus a need to develop a distributed setup PSA scheme which is *efficient* and *proven secure.*

1.1 Contributions

In this paper we (1) introduce a definition and a security model for a *distributed setup* PSA scheme, (2) present DIPSAUCE, the first PSA scheme which does not rely on a trusted party, (3) prove this scheme secure under static corruptions (and sketch modifications for security under mobile corruptions), (4) show its practical feasibility 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 (5) implement two state-of-the-art PSA schemes [21, 45] on the same devices and compare the performance to our scheme. All code is available at [11], and the raw measurement data are available at [10].

DIPSAUCE is defined and proved in the standard model. Our implementation, however, uses a more practical building block with a hash function assumed to be a random oracle for better performance. Note that this is implementation specific rather than a limitation of the protocol.

Looking ahead, when comparing the setup and keygen procedures in DIPSAUCE with the suggestions for a distributed setup in KH-PRF-PSA [21] and LaSS-PSA [45] with 10000 parties, our results show a speedup of 78x and 49x respectively. For the encryption procedure our protocol shows a speedup of 22x compared to KH-PRF-PSA and 50x compared to LaSS-PSA.

1.2 Our Techniques

Our PSA scheme DIPSAUCE is a variant of LaSS-PSA, and takes inspiration from the sketches of a distributed setup proposed in [21, 45]. The security of these proposed distributed setups crucially relies on each party deriving a shared key for *each other party*. This approach is secure against an *adaptive* adversary allowed to corrupt up to $n-2$ out of all n parties, but not the targeted device itself. As we show later, the $O(n)$ complexity of this technique makes such a protocol infeasible on constrained devices for the number of parties (1000-10000) suggested in [21, 45].

To avoid this situation, we first consider a *static* adversary instead, which can corrupt up to t out of the n parties (where *e.g.* $t = \frac{n}{2}$) before the start of the protocol execution. This allows us to design a more efficient protocol using the technique of *hidden committees* [9, 16]. Specifically, we leverage a *randomness beacon* to determine a small random committee for each user, consisting of k, $(k \ll n)$ other parties to derive a shared key with. Such a committee (probabilistically) preserves the ratio of corrupted parties [19], and prevents a static adversary from targeting committee members for corruption. Thus, the complexity can be reduced from $O(n)$ to $O(k)$ $(k \ll t < n)$, while maintaining security, so that the setup stage is efficient enough to be feasible.

We then adapt our protocol to be secure under a stronger security model where the adversary can corrupt parties throughout the execution of the protocol. A standard notion of such an adversary is the mobile adversary model, where the adversary periodically corrupts a new set of up to t parties while retaining knowledge of any learned secrets from the previously corrupt parties. It can thus corrupt all parties over time, just not more than t during the same time period. A secure protocol must thus *heal* the parties whose secrets were leaked to the adversary. We, therefore, present an extension of our protocol which protects against a mobile adversary by periodically erasing certain secrets and then re-running the distributed setup to create fresh ones. This approach is practically possible due to the efficiency of the distributed setup in DIPSAUCE.

1.3 Outline

The rest of this paper proceeds as follows. In Section 2, we introduce notation and recall definitions of well known schemes. Then, we evaluate the efficiency of the state of the art PSA schemes in Section 3, and evaluate the efficiency of their proposed methods for distributing their setup procedures in Section 4. In light of these results, we then propose our novel PSA scheme and prove it secure in Section 5, and evaluate its efficiency and compare it to the state of the art schemes in Section 6. Finally, we conclude and discuss some practical properties of our protocol in Section 7.

2 PRELIMINARIES

In this section we introduce notation and recall known constructions relevant to our scheme.

Notation. Throughout the paper $\lambda \in \mathbb{N}$ denotes the computational security parameter. A specific party in a scheme is denoted as $\mathcal{P}_i$. We will 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 $\{1, \ldots, n\}$. We denote the set of permutations of $[n]$ by $\mathsf{Perm}(n)$ and the k:th permutation of this set as $\mathsf{Perm}_k(n)$. For a permutation of $[n]$, $\rho_k = \mathsf{Perm}_k(n)$, we denote the value of i:th element in ρ_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 $N(v_i)$. We also let $\vec{J}_i$ denote 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. A Private Stream Aggregation (PSA) scheme is a scheme which for each label l allows an evaluator to learn the sum of inputs $\{m_1, \ldots, m_n\}$, from a set of parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ without learning the individual inputs.

In existing PSA schemes, a trusted party executes the setup procedure and distributes the secret keys to the aggregator and clients. We here recall the definition of such centralized setup PSA schemes and their corresponding security notion of *aggregator obliviousness*, as defined in [21].

Definition 1 (Private Stream Aggregation). *A PSA scheme is defined by the procedures:*

- $\mathsf{Setup}(\lambda, n)$*: On input the security parameter* λ *and the number of parties* n*, output public parameters* pp *and* $n+1$ *secret symmetric encryption keys* $\{\mathsf{ek}_i\}_{i \in [n+1]}$ *(where* ek_{n+1} *is the aggregator key, sometimes alternatively denoted as* ek_a*).*
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, m_i, l)$*: On input the public parameters* pp*, an encryption key* ek_i*,* $i \leq n$*, a message* $m_i \in \mathbb{Z}_R, R \in \mathbb{N}$ *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* pp*, the aggregator key* 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 PSA scheme PSA = (Setup, Enc, AggrDec) must satisfy *correctness*. 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 $\forall \{c_i\}_{i\in[n]} : c_i = \mathsf{Enc}(\mathsf{pp}, \mathsf{ek}_i, l, m_i)$, correctness is satisfied if:

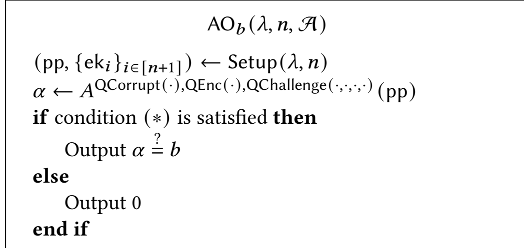
$$\mathsf{AggrDec}(\mathsf{pp}, \mathsf{ek}_a, l, \{c_i\}_{i\in[n]}) = \sum_{i\in[n]} m_i \pmod R$$

Further, a *secure* 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 2 (Aggregator Obliviousness). *Let* PSA *be a* PSA *scheme. Let the experiment AO_b in Figure 1 be defined with the following oracles:*

- $\mathsf{QCorrupt}(i)$*: The oracle outputs the encryption key* ek_i *of user i. For $i = n + 1$, it outputs the aggregator key* 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, $\mathcal{A}$ outputs a guess α, whether $b = 0$ or $b = 1$. A PSA scheme is Aggregator Oblivious, *if for every PPT adversary $\mathcal{A}$ there exists a negligible function* negl *such that for all sufficiently large λ, $\mathsf{Adv}^{AO}_{\mathcal{A},\mathsf{PSA}}(\lambda, n) = \mathsf{negl}(\lambda)$.*


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

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

Figure 1: The aggregator obliviousness game defining security for a PSA scheme.

To formally define the condition $(*)$, the following sets are introduced:

- 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 1), 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^*). Doing so would violate the encrypt-once restriction.
- $\mathcal{U} \cap \mathcal{E}_{l^*} = \emptyset$. 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.*

In [45] and [21] the authors consider adaptive corruptions. $\mathcal{A}$'s advantage is defined as

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

k-Regular Graphs. A k-regular graph is a graph in which each vertex has exactly k neighbours. Efficient algorithms for generating regular graphs are well known [34].

Randomness Beacons. A beacon [6] is a function $r = \mathsf{Beacon}(t)$ which returns an m-bit near-uniformly random value r at each time interval t. Informally, a secure beacon should be *unpredictable*, *i.e.* the advantage for an adversary predicting r before time t should be negligible, *unbiased*, *i.e.* r is statistically close to an m-bit uniformly random string, *universally samplable*, *i.e.* any party should be able to obtain r after time t, and *universally verifiable i.e.* any party can verify that no party had access to the random sample used to construct r before time t.

Non-interactive Key Exchange. We here recall Non-Interactive Key Exchange (NIKE) as defined in [18].

Definition 3 (Non-Interactive Key Exchange). *A scheme for Non-Interactive Key Exchange consists of the following algorithms:*

$\mathsf{Setup}(\lambda)$*: On input a security parameter λ, output public parameters* pp.

$\mathsf{KeyGen}(\mathsf{pp})$*: On input the public parameters* pp*, output a party's public key* pk_i *and corresponding secret key* sk_i.

$\mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_j)$*: On input the public parameters* pp*, a public key* pk_i *and secret key* sk_j*, deterministically output a shared key K.*

We say that a NIKE scheme is correct if $\Pr[\mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_i, \mathsf{sk}_j) = \mathsf{SharedKey}(\mathsf{pp}, \mathsf{pk}_j, \mathsf{sk}_i)] = 1$. We say that a 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 from a random string of the same length, when both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest. We refer to [18, Def. 15] for a full definition of the security game.

Pseudo Random Functions. Let $\mathcal{F}$ denote a family of efficiently-computable functions $F_k : K \times X \rightarrow Y$ indexed by $k \in K$. The family $\mathcal{F}$ is said to be a (t, ϵ) strong PRF if for every $k \in K$, and all adversaries $\mathcal{A}$ running in time t can not distinguish $F(k, .)$ from a random function $f : X \rightarrow Y$. To be formal, we write:

$$|Pr[\mathcal{A}^{F_k(.)} = 1] - Pr[\mathcal{A}^{f(.)} = 1]| < \epsilon$$

To aid the reader, we will denote such a function F_k PRF_k from here on. $(\mathcal{F})_{\mathsf{KH\text{-}PRF}} \subset \mathcal{F}$ is said to be *additively* key-homomorphic if for $F_{k_i}, F_{k_j} \in (\mathcal{F})_{\mathsf{KH\text{-}PRF}}$ the following condition holds

$$F_{k_i}(x) + F_{k_j}(x) = F_{k_i+k_j}(x)$$

We will denote such a function $\mathsf{KH\text{-}PRF}_k$.

3 EVALUATING THE PERFORMANCE OF STATE OF THE ART PSA SCHEMES

In this section, we evaluate the performance of the client side operations (*i.e.* the encryption procedure, since the schemes are defined with a centralized setup) of the PSA schemes in [21] and [45], which can be considered the current state of the art due to them being the most efficient schemes.

We note that there exists a recent scheme [43] which measures faster encryption times than [21] and [45]. However, these results only measure the "on-line" time and do not include the precomputations which they rely upon. While such precomputations are interesting to explore, and indeed could be done for both [21] and [45], a deeper study taking trade-offs such as time vs. memory etc. into account is needed in order to warrant a direct comparison. We therefore consider [21] and [45] to be the state of the art for this work.

The first evaluated scheme [21], which we will call KH-PRF-PSA is based on a Key Homomorphic PRF. The second scheme [45], which we refer to as LaSS-PSA is based upon a primitive called Labeled Secret sharing (LaSS), which is also introduced in [45].

Both these works evaluate the performance of their schemes on Intel i5 CPUs, rather than on constrained devices. Therefore, these performance evaluations give little insight into how the schemes perform on realistic hardware. We therefore investigate whether these schemes can be run on constrained devices. The code for our experiments and protocol implementation is available at [11]. The raw data of the results are available at [10].

3.1 Scenario and Experiment Setup

Scenario. In our IoT data analytics scenario, a set of *n Clients*, each measures some statistic, *e.g.* power in a smart metering scenario, and wishes to communicates the sum of the measurements to a *Server* (without revealing individual measurements).

Setup. We evaluate the client side operations of PSA schemes on devices from the CC1352 series of devices with ARM Cortex M4 processors. These devices can be considered “mid-range" constrained devices, as they are classified as C3 devices in the IETF draft for terminology on constrained devices [8]. The CC1352 platform has hardware acceleration for Elliptic Curve Diffie-Hellman, AES and SHA256, as well as a True Random Number Generator (TRNG) and a capabilities to run the wireless protocol IEEE802.15.4 Low Power Personal Area Network (LoWPAN). The computer running the *Server*, *i.e.* the aggregating party, has a *Border-Router* with IEEE802.15.4 capabilities that connects to the *Client*. We provide additional details on the experiment setup and the CC1352 platform in Appendix A.

Experiments. We evaluate the client side efficiency of LaSS-PSA and KH-PRF-PSA by measuring the execution time of the respective Enc procedures. Time is measured from the start of the process until the ciphertext is ready to be transmitted. No network operations are included in this test. In the experiments in [21, 45], 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 the experiments in previous works, 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 6.1. The encrypt procedure was repeated 10 times for each group size, and our results are the average of these 10 runs.

3.2 Protocol Definitions and Implementations

3.2.1 KH-PRF-PSA.

Let us recall the definition of the KH-PRF-PSA protocol from [21] in Protocol 1.

Protocol 1 – The KH-PRF-PSA scheme in [21].

$\mathsf{Setup}(\lambda, n)$:

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** $\mathsf{ek}_a, \{\mathsf{ek}_i\}_{i\in[n]}$

$\mathsf{Enc}(\mathsf{ek}_i, m_i, l)$:

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

$\mathsf{Aggr}(\mathsf{ek}_a, \{c_i\}_{i\in[n]}, l)$:

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 [21, Sec. 4], which uses a KH-PRF based on a hash function $H()$, and which is secure in the Random Oracle Model (ROM). 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 λ elements from $\mathbb{Z}_q$ and thus has size $\lambda \cdot q = 33536$ 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):

$$\mathsf{KH\text{-}PRF}_{\mathsf{ek}}(l) = \lfloor \langle H'(l), \mathsf{ek} \rangle \rfloor_p \quad (1)$$

This definition uses the syntax of [21], where: $\lfloor x \rfloor_p = \lfloor x \cdot p/q \rfloor$. The function $H'(l)$ is in turn defined as a vector of λ hashes of the label concatenated with a counter, and reduced modulo q:

$$H'(l) = \begin{pmatrix} H(l||""||"1") \pmod q \\ \vdots \\ H(l||""||"\lambda") \pmod q \end{pmatrix} \quad (2)$$

In the instantiation in [21] SHA3-512 is used for the hash function $H()$. For a fair comparison, we however select a more efficient hash function, SHA256, which is hardware accelerated on the CC1352 platform.

3.2.2 LaSS-PSA. Let us also recall the LaSS-PSA scheme from [45], presented in Protocol 2. 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.

Protocol 2 – The LaSS-PSA scheme [45].

Setup(λ, n):

1: $\forall i \in [n+1], \forall j$ s.t. $n \le j > i : K_{i,j} \xleftarrow{\$} \mathbb{Z}_R$
2: $\forall i \in [n+1], \forall j$ s.t. $n \le j < i : K_{i,j} = K_{j,i}$
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

We here implement the version which instantiates the PRF using AES-128, since its the most efficient instantiation of LaSS in the measurements of [45], and is hardware accelerated on the CC1352 platform. The encryption key ek consists of a vector of n elements from $\mathbf{Z}_R$, where n is the number of users in the system, and thus has size $n \cdot log_2(R)$.

3.3 Results

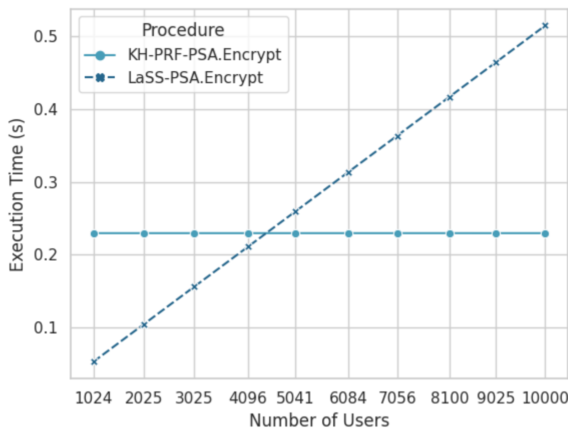


Figure 2: Execution time in seconds of the Enc **procedure in** KH-PRF-PSA **and** LaSS-PSA**.**

The results of the experiments are available in Figure 2. 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, regardless of the numbers of parties. 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 [21, 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 [21]. This discrepancy stems from the fact that our experiments are executed on a constrained devices whereas the experiments in [21] are executed on an Intel Core i5 CPU.

4 EVALUATING THE METHODS FOR A DISTRIBUTED SETUP PROPOSED IN [21, 45]

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 [21] and LaSS-PSA [45], 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. In this section, we 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 with an ARM Cortex M4 processor, and evaluate the performance of the client side operations (*i.e.* both setup and encryption since the setup is now performed by the clients instead of by a trusted party). This gives us an estimate for the performance of this approach to distributing the setup of PSA scheme. The code for our experiments and protocol implementation is available at [11]. The raw data of the results are available at [10].

It is not our intent to prove the security of this distributed setup procedures. We here only wish to show its (in)efficiency. Definitions and proofs for distributed setup are instead available for the scheme in Section 5.

4.1 A Distributed Setup for KH-PRF-PSA

Ernst and Koch propose a decentralized setup protocol in [21, Section 5.1] based on the sum-of-PRF technique. Let us here briefly recall this technique.

*Sum-of-*PRF*s.* The sum-of-PRFs technique, first introduced as part of a scheme in [15] and later used in *e.g.* [18, Sec. 6.2], allows the parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ to derive a sum of their respective 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 from Section 2). 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.

Decentralizing the Protocol. Protocol 3 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 1, the evaluator must derive k_a as the sum of all aks_i, *i.e.* $k_a = \sum_{i \in [n]} \mathsf{aks}_i$.

Protocol 3 – Distributed Setup for KH-PRF-PSA

Setup(λ, n, k, 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 pk_j for each $j \in [n]$
7: **for** $j \in [n] \setminus \{i\}$ **do**
8: $\quad K_j \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_j, \mathsf{sk}_i)$
9: **end for**
10: **for** $\ell \in \{1, \ldots, \lambda\}$ **do**
11: $\quad b_{i,\ell} \leftarrow \sum_{j \in [n] \setminus \{i\}} (-1)^{i<j} \cdot \mathsf{PRF}_{K_j}(\ell)$
12: $\quad \vec{A}_i[\ell] = \vec{E}_i[\ell] + b_{i,\ell} \pmod R$
13: **end for**
14: $aks_i \leftarrow \vec{A}_i[1] || \ldots || \vec{A}_i[\lambda]$
15: **return** $\mathsf{ek}_i, \mathsf{aks}_i$

In our implementation of Protocol 3, we use a python based PKI with a CoAP [40] interface where all keys of other parties are registered. We have instantiated NIKE using ECDH P-256. ECDH P-256 is hardware accelerated on the CC1352 platform. The PRF was instantiated using hardware accelerated AES-128.

4.2 A Distributed Setup for LaSS-PSA

Waldner et al. propose a decentralized setup protocol in [45, Sec. 7], which we give details on how to construct in Protocol 4. Note that to make this setup compatible with Protocol 2, the aggregator must also execute the setup protocol.

Protocol 4 – Distributed Setup for LaSS-PSA.

Setup(λ, k, 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 pk_j for each $\mathcal{P}_j \in \mathcal{P}$
5: **for** $j \in [n] \setminus \{i\}$ **do**
6: $\quad K_j \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_j, \mathsf{sk}_i)$
7: **end for**
8: **return** $\mathsf{ek}_i = \vec{K}_i$

In our implementation of Protocol 4, we use a python based PKI with a CoAP [40] interface where all keys of other parties are registered. We have instantiated NIKE using ECDH P-256. ECDH P-256 is hardware accelerated on the CC1352 platform. The PRF was instantiated using hardware accelerated AES-128.

4.3 Experiments and Results

The setup and experiments described in this section is the same as described in Section 3.1 and Appendix A, however instead of measuring Enc, we measure the Setup execution times. The results of the experiments are available in Figure 3.

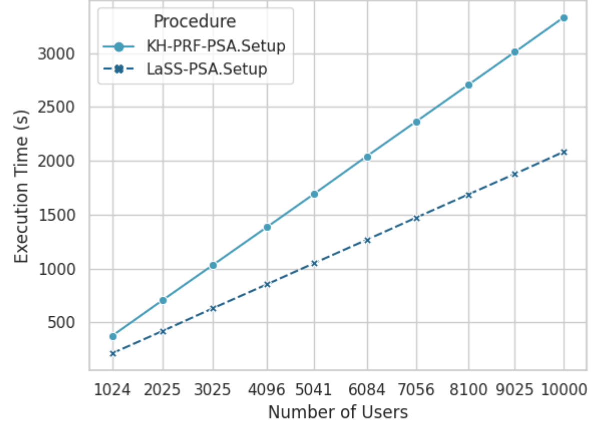


Figure 3: Execution time in seconds of the Setup **procedure in** KH-PRF-PSA **and** LaSS-PSA**.**

The figure shows that the execution times of the setup parts of both 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 3 and Protocol 4), also generates a larger secret key ek_i (steps 1-2 in Protocol 3) and masks ek_i before it is sent to the aggregator (steps 10-14 in Protocol 3). Thus LaSS-PSA outperforms KH-PRF-PSA for all number of users in the system.

5 EFFICIENT DISTRIBUTED SETUP PSA

The results in the previous section show that the existing suggestions for obtaining a distributed setup for PSA schemes in [21, 45] are infeasible in practice. To address this, we now present our protocol DIPSAUCE. It can be seen as a distributed setup variation of the protocol in [45, Section 4], modified so that the Setup procedure no longer generates keys centrally. Instead, we introduce a KeyGen procedure which each party executes independently.

Although we here present a specific protocol, our approach can be used to distribute the setup of other PSA schemes, for example the scheme in [21].

Approach. The distributed setup procedures in Section 4 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 targeted device is secure against an *adaptive* adversary which corrupts up to $n-2$ of the committee parties (but not the targeted device itself). While this is a very strong security guarantee, we have shown in previous sections that the resulting protocol is rendered too inefficient for practical use. The main bottleneck giving rise to this inefficiency is the size of the committee.

How then to enable more efficient constructions by reducing the size of the committee, without sacrificing security? For example a committee of size $\sqrt{n}$ would be much more efficient, but if it is

only secure against up to $\sqrt{n}-1$ corruptions it cannot be said to be as secure. 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 the protocol lays in how it uses a k-regular graph and a randomness beacon to efficiently establish unpredictable committees. The protocol defines each committee using the output of a public randomness beacon. However, an efficient protocol cannot directly use the output of the beacon to determine the committees. Sampling n committees of size k and transferring this data to the devices would mean transferring nk group elements to each device, which is not feasible in scenarios with 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 k other parties.

Aggregation output. In line with [21, 45], 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 specifically allow only one particular aggregator party to aggregate, then this property can be obtained by sending the ciphertexts over an encrypted channel to the aggregator party, or alternatively by including the aggregator among the encrypting parties and letting it encrypt zero without publishing the ciphertext.

5.1 Definition

Assumptions. We assume that all parties have access to a Randomness Beacon (RB) and a Public Key Infrastructure (PKI). We presuppose that each node in G is assigned an index to indicate its corresponding vertex in the graph during setup.

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 full control of the execution of that party, meaning that it controls the actions and learns the internal state of a corrupt party throughout the execution of the protocol. The set of corrupt parties is denoted C.

Definition 4 (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 $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ and a special party called the evaluator $\mathcal{E}$, and consists of the following procedures:*

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

$$\mathsf{AO}_b(\lambda, n, \mathcal{A})$$
$$\begin{array}{l} L \leftarrow \emptyset \\ \mathsf{pp} \leftarrow \mathsf{Setup}(\lambda, conf) \\ \textbf{for } i \in [n] \textbf{ do} \\ \quad \mathsf{ek}_i \leftarrow \mathsf{KeyGen}(\mathsf{pp}, i) \\ \textbf{end for} \\ \gamma \leftarrow \mathcal{A}^{\mathsf{QEnc},\mathsf{QLeftRight}} \\ \textbf{return } \gamma \stackrel{?}{=} b \end{array}$$

Figure 4: The aggregator obliviousness experiment defining security for a distributed setup PSA scheme.

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:

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

A DS-PSA scheme is secure if an adversary has a negligible probability of winning the game for Aggregator Obliviousness (AO) in Definition 5.

Definition 5 (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 4. $\mathcal{A}$ 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 $(i, l^*) \notin L$ then it lets $L \leftarrow L \cup \{(i, l^*)\}$ and answers the query with $c_i = \mathsf{Enc}(\mathsf{ek}_i, m_i, l^*)$.*
- $\mathsf{QLeftRight}(\mathcal{U}, \{m_i^0\}_{i\in\mathcal{U}}, \{m_i^1\}_{i\in\mathcal{U}}, l^*)$*: 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 $\{\mathcal{P}_i\}_{i\in\mathcal{U}} \cap C = \emptyset$ and no previous calls has been made to* QLeftRight*. If further $\{\mathcal{P}_i\}_{i\in\mathcal{U}} \cup C = \{\mathcal{P}_i\}_{i\in[n]}$ 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 answers the query with $\{c_i\}_{i\in\mathcal{U}}$ s.t. $c_i = \mathsf{Enc}(\mathsf{ek}_i, m_i^b, l^*)$.*

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

This security definition models *encrypt-once* security, i.e. the restriction that each party only encrypts a single message per label (which is the natural usage of the scheme). This is enforced by both QEnc and QLeftRight maintaining the set L, where they store which label has been used for each user and ignores any requests of label reuse. Further, since any party has the ability to aggregate in Definition 4, 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 $\{m_i^1\}_{i\in\mathcal{U}}$.

We note that this AO-game is similar to the AO-games in [21, 45]. The main differences are that we model corruptions as a full party takeover rather than as a key leaking oracle, and the lack of a dedicated key for the aggregator.

5.2 Construction

The protocol is described in Protocol 5.

Protocol 5 – DIPSAUCE

Setup($\lambda, conf = \{n, k, time\}$):

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

KeyGen(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** $\ell \in \{1, \dots, k\}$ **do**
7: $\quad \ell' = \vec{J_i}[\ell]$
8: $\quad$ Wait until the PKI returns an entry $\mathsf{pk}_{\ell'}$ for $\mathcal{P}_{\ell'}$
9: $\quad \vec{K_i}[\ell] \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{pk}_{\ell'}, \mathsf{sk}_i)$
10: **end for**
11: **return** $\mathsf{ek}_i = (\vec{K_i}, \vec{J_i})$

Enc(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: $c_i = (t_i + m_i) \pmod R$
3: **return** c_i

Aggr(pp, $\{c_i\}_{i\in[n]}$):

1: $M = \sum_{i\in n} c_i \pmod R$
2: **return** M

Correctness. By definition we have

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

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

$$i \in \vec{J}_{i_j} \text{ for } j = 1, \dots, k.$$

For simplicity we let i' denote any one of the indices i_j above. Furthermore, 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 DIPSAUCE is correct as long as NIKE is correct and G is k-regular, since then all $K_i[\ell]$ will cancel out during aggregation s.t. $\sum_{i\in n} t_i = 0$.

Game	Definition of QLeftRight-oracle	Argument
G_0	$t_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J_i}[\ell]} \cdot \mathsf{PRF}_{\vec{K_i}[\ell]}(l)$ $c_i \leftarrow m_i^0 + t_i$	
G_1	$\boxed{t_i' \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J_i}[\ell]} \cdot \mathsf{PRF}_{\vec{K_i}[\ell]}(l)}$ $\boxed{t_i \leftarrow t_i' + \mathsf{PSS}(0, i, n - \lvert C\rvert)}$ $c_i \leftarrow m_i^0 + t_i$	t_i indisting. from rand.
G_2	$t_i' \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J_i}[\ell]} \cdot \mathsf{PRF}_{\vec{K_i}[\ell]}(l)$ $t_i \leftarrow t_i' + \mathsf{PSS}(0, i, n - \lvert C\rvert)$ $c_i \leftarrow \boxed{m_i^1} + t_i$	one-time-pad info. theo. secure
G_3	$\boxed{t_i \leftarrow \sum_{\ell=1}^{k} (-1)^{i<\vec{J_i}[\ell]} \cdot \mathsf{PRF}_{\vec{K_i}[\ell]}(l)}$ $c_i \leftarrow m_i^1 + t_i$	t_i indisting. from rand.

Table 1: Strategy for proving AO-Security of DIPSAUCE. The change in each game is highlighted by boxing.

5.3 Security Analysis

Since DIPSAUCE is a variation of [45], it can be proven using the same proof strategy (originating from [1] and also used in [21]), which consists of a series of games forming a hybrid argument, and where each game changes the definition of the QLeftRight-oracle. We recall this strategy in Table 1.

The game G_0 corresponds to the AO_0-game where QLeftRight queries are answered with the encryption of m_i^0. G_3 corresponds to the AO_1-game where 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 AO_0-game from the AO_1-game.

The transition from G_0 to G_1 consists of adding a *perfect* secret sharing (denoted 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. Thus 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 G_1 to G_2, where c_i now encrypts m_i^1 instead of m_i^0. This transition is justified since t_i is 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 G_2 to G_3 consists of undoing the change made in G_1 (with the same security argument). We arrive at the following theorem.

Theorem 5.1. *If t_i is indistinguishable from randomness for a computationally bounded adversary except with a negligible advantage, then* DIPSAUCE *is* AO-*secure.*

5.3.1 *Proving the threshold sum-of-PRFs technique.*

We now prove that t_i is indistinguishable from randomness in the presence of a

static malicious adversary, *i.e.* an adversary limited to corrupting parties only *before* the start of protocol execution. In DIPSAUCE t_i is generated with a threshold version of the Sum-of-PRFs technique, where for each $\mathcal{P}_j$, the selection of which PRFs to include in its sum is determined by its k-sized committee, equal to the set of neighbours to the users vertex in a random permutation of the graph G.

Proof Outline. The outline of the proof is as follows. We first formalize the security of our known building blocks of NIKE and sum-of-PRFs in the context of our scheme in Lemma 5.2 and Lemma 5.3. Intuitively Lemma 5.2 states that all NIKE derived keys are private to the negotiating parties, and Lemma 5.3 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 are then ready to consider the DIPSAUCE method, where k-sized committees are selected at random from a population of n parties with a threshold t of corrupt parties. This is formalized in Theorem 5.4.

We then conclude by formalizing the indistinguishably of t_i as a consequence of the previous theorem and lemmas in Theorem 5.5.

Details of the proof. Let us now detail the different parts, beginning with restating the security of NIKE in the context of our scheme, *i.e.* that each NIKE derived key derived for a committee member, is does not leak anything to the adversary for all *honest* parties in the users committee. As a direct consequence of the security of NIKE, Lemma 5.2 is true.

Lemma 5.2 (Pseudo-Random Shared Keys). DIPSAUCE.KeyGen *outputs* $\mathsf{ek}_i = (\vec{K}_i, \vec{J}_i)$ *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.*

Let us also briefly 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 for the label l. We get *Lemma* 5.3.

Lemma 5.3 (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 5.3, we can only say that the protocol is secure against an adversary corrupting up to $t = k-1$ parties. Let us therefore transfer from the standard sum-of-PRFs technique to our threshold version.

Theorem 5.4 informally states that if we randomize the committee members, an adversary corrupting up to t parties will have a negligible chance to corrupt *all* k committee members of a user with these t corruptions.

In the proof of this theorem, we first argue that the permutation of the graph is pseudorandom.

Then, as a stepping stone, we first consider the advantage of the adversary in guessing a *specific* random committee. Intuitively, if we randomize the committees for each user, a static adversary has no better strategy than to randomly guess the k users in the committee. To then 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, we then finally consider an adversary which attempts to learn *any* t_i.

Theorem 5.4 (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. The permutation ρ 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 ρ except with a negligible advantage. Then, since $|G| = |\rho|$, the adversary has a negligible advantage in determining which $\mathcal{P}_i$ is associated with which $v_j \in G$.

Consider the number of possible k-sized committees and the number of k-sized committees an adversary can form from t random corruptions. The total number of possible 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 $\frac{\binom{t}{k}}{\binom{n}{k}}$.

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 $n \cdot \frac{\binom{t}{k}}{\binom{n}{k}}$.

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 Section 5.5 for a discussion on the values of n, t, k).

□

Now, since a static adversary cannot corrupt all nodes in the committee of any honest party (Theorem 5.4), and the sum-of-PRFs technique is secure if there is at least one honest committee member (Lemma 5.3), t_i is indistinguishable from randomness.

Theorem 5.5 (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.*

5.4 Proactively secure DIPSAUCE

The above section proves the DIPSAUCE protocol secure in the static security setting. In this section we will sketch how to construct a proactively secure version of DIPSAUCE, *i.e.* a version secure in the presence of a *mobile* adversary. Trivially, by re-running the Setup and KeyGen procedures at the beginning of each epoch, proactive security is achieved. Since the Setup and KeyGen procedures are efficient in DIPSAUCE, this modification is feasible in practice.

5.4.1 Modelling proactive security. We model the proactive security property according to [38], allowing corruptions and uncorruptions at epoch changes as follows.

Epochs Time is divided into consecutive *epochs*, where each epoch is indexed by an incrementing epoch counter.

Corruptions A mobile adversary $\mathcal{A}$ 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 it previously learned from that party, but has no further control of the execution of that party and learns no further secrets. The total number of corrupt parties at the start of an epoch can never exceed t. As a consequence, all parties can be corrupt during some stage of the protocol execution, but the adversary learns secrets from at most t parties during each individual epoch.

5.4.2 Achieving proactive security for DIPSAUCE. By discarding all secrets and starting an epoch with fresh secrets, we can achieve proactive security. For brevity, we have so far omitted how the PKI trust relation is achieved, *i.e.* how the PKI verifies that a posted public key actually belongs to the claimed identity. The caveat to discarding all secrets is how to maintain this PKI trust relation, in order to to prevent impersonations, over epochs. This problem has been studied in the literature before [38].Let us go into some detail of the known solutions.

When the adversary leaves a party, it still retains all variables learnt during corruptions, including any secret used to establish the trust relation with the PKI. Thus, in the mobile scenario, we must additionally prevent the adversary from using this knowledge to impersonate previously corrupt parties, during subsequent epochs whenever the party is honest. Otherwise, another honest party might derive a shared key by using a public key posted to the PKI by the adversary, believing it to be the public key of an honest party. When secrets are deleted at the end of an epoch, this includes any secret related to the trust relation with the PKI, and the trust relation is then destroyed. The challenge of achieving proactive security for DIPSAUCE thus hinges on maintaining a trust relation with the PKI in between epoch changes.

In [38] two methods of maintaining such trust relations are described. 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) [23] or trusted execution techniques that provide secure storage [39] for a PKI relation root key.

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. An adversary can of course also post a new key signed with the previous key. However, in that case, since an honest party will also post a new key, the system will notice that two public keys have been published, signed with the same secret key. The system can then deduce that the corresponding device has been compromised. This assumes that the adversary cannot suppress legitimate messages reaching their destination.

Details on how to implement this in DIPSAUCE. We 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.

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.

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 pp as output of 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 informal theorem:

THEOREM 5.6 (INFORMAL). *Let there be a scheme such 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.*

5.5 Parameter Selection for n, t and k

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 5.4. 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 of is available in Appendix B.

6 EXPERIMENTAL EVALUATION

In this section we describe our implementation of DIPSAUCE and perform a comparative evaluation of DIPSAUCE against the state of the art protocols KH-PRF-PSA and LaSS-PSA modified to utilize distributed setup as described in Section 4. The code for our experiments and protocol implementation is available at [11]. The raw data of the results are available at [10].

6.1 Implementation of DIPSAUCE

Let us now describe our implementation of Protocol 5. The Setup procedure is hard-coded into the source code. Here 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 generated [34]. The KeyGen procedure is straightforwardly implemented according to Protocol 5, using a python based PKI with a CoAP [40] interface where all keys of other parties are registered, using the Drand public randomness beacon [37], and instantiating NIKE as ECDSA on the P-256 curve. We have implemented the Enc procedure by instantiating the PRF using AES-128. Both AES-128 and ECDSA P-256 utilizes the hardware acceleration of the CC1352 platform.

6.2 Experimental Setup

We have used the same experimental setup as described in Section 3.1 with the same suite of experiments, *i.e.* measuring the setup, (including keygen for DIPSAUCE) and encrypt procedures described in Section 3.1 and Section 4.3.

Execution times for the setup procedure are measured from the start of the process, including the time needed to transfer data, such as keys, over the network. For the encryption procedure, execution times excludes the time needed to transfer the encrypted message.

6.3 Results

6.3.1 Setup and KeyGen. Our evaluation shows that DIPSAUCE significantly outperforms both KH-PRF-PSA and LaSS-PSA in terms of execution time for the setup (and keygen) procedure. We show a plot of the execution times of these procedures in Figure 5. The slope of the graphs mean that DIPSAUCE will have the shortest execution time of the protocols 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, as explained in Section 4.3, being more efficient than KH-PRF-PSA. Compared to LaSS-PSA, our protocol DIPSAUCE shows a speedup of 66x.

6.3.2 Encrypt. Our evaluation of the Enc procedures show 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 6. LaSS-PSA and DIPSAUCE show a linear performance, depending on 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 eventually outperform DIPSAUCE. Extrapolating from the measured execution times, this will be the case when $n \approx 300000$.

7 DISCUSSION

In this paper we have evaluated two state-of-the-art PSA schemes that rely on a *centralized* setup. We have experimentally evaluated proposed ways to bypass the centralized setup, but found the solutions infeasible in a practical environment. The reason for this is the 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 points.

Client Failures. If a single ciphertext from an honest client is missing at the aggregator, the security definition of a PSA scheme requires that the aggregator learns nothing. This is the point of a PSA scheme and considered a feature. However this feature can be a problem in practice if ciphertexts are lost due to client failures. This practical problem is dealt with in [14], which proposes a general solution for dealing with client errors and which is applicable to all PSA schemes including ours. Since the setup in DIPSAUCE is

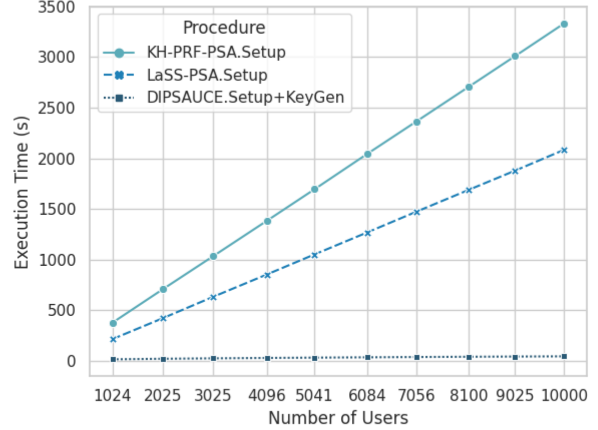


Figure 5: Execution time in seconds of the Setup procedure of KH-PRF-PSA and LaSS-PSA and the Setup and KeyGen procedure of DIPSAUCE

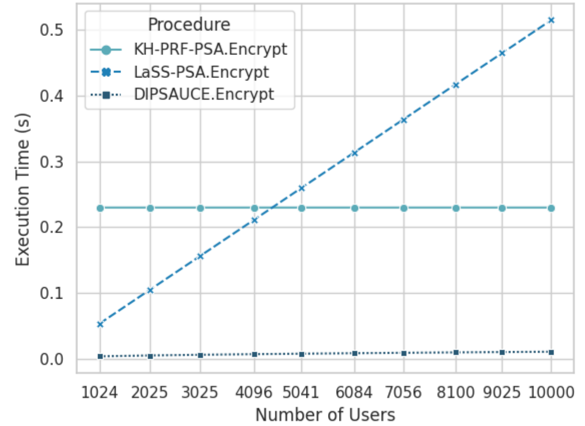


Figure 6: Execution time in seconds of the Encrypt procedure of of KH-PRF-PSA, LaSS-PSA, and DIPSAUCE.

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.

Relying on an external PKI and Randomness Beacon. The distributed KeyGen procedure in DIPSAUCE relies on a PKI to supply the correct public key for each user, which is often implemented as a central entity. While this is a standard assumption, we note that it is possible to *distributively audit* a PKI for correct behaviour [29, 32, 33].

Analogously, DIPSAUCE relies on a randomness beacon. We therefore remark that one must be careful when realizing the beacon, in order to not introduce a trusted party. Multiple solutions for beacons which do not rely on a trusted party exist, for example beacons based on multiparty randomness generation protocols [12] or beacons utilizing the existing distributed security of Bitcoin [6].

ACKNOWLEDGMENTS

We would like to thank Paul Stankovski Wagner and Elena Pagnin for valuable discussions of this work. Joakim Brorsson was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Martin Gunnarsson was supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research.

REFERENCES

[1] Michel Abdalla, Fabrice Benhamouda, and Romain Gay. 2019. From single-input to multi-client inner-product functional encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, Berlin, Germany, 552–582.

[2] Michel Abdalla, Fabrice Benhamouda, Markulf Kohlweiss, and Hendrik Waldner. 2019. Decentralizing inner-product functional encryption. In *IACR International Workshop on Public Key Cryptography*. Springer, Berlin, Germany, 128–157.

[3] Daniela Becker, Jorge Guajardo, and Karl-Heinz Zimmermann. 2018. Revisiting Private Stream Aggregation: Lattice-Based PSA.. In *NDSS*. Internet Society, Reston, VA, USA, 17 pages.

[4] Fabrice Benhamouda, Marc Joye, and Benoît Libert. 2016. A new framework for privacy-preserving aggregation of time-series data. *ACM Transactions on Information and System Security (TISSEC)* 18, 3 (2016), 1–21.

[5] Dan Boneh, Amit Sahai, and Brent Waters. 2011. Functional encryption: Definitions and challenges. In *Theory of Cryptography Conference*. Springer, Berlin, Germany, 253–273.

[6] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. 2015. On Bitcoin as a public randomness source. Cryptology ePrint Archive, Paper 2015/1015. https://eprint.iacr.org/2015/1015 https://eprint.iacr.org/2015/1015.

[7] C. Bormann, M. Ersue, and A. Keranen. 2014. *Terminology for Constrained-Node Networks*. RFC 7228. RFC Editor. http://www.rfc-editor.org/rfc/rfc7228.txt http://www.rfc-editor.org/rfc/rfc7228.txt.

[8] Carsten Bormann, Mehmet Ersue, Ari Keränen, and Carles Gomez. 2022. *Terminology for Constrained-Node Networks*. Internet-Draft draft-ietf-lwig-7228bis-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-lwig-7228bis/00/ Work in Progress.

[9] Joakim Brorsson, Bernardo David, Lorenzo Gentile, Elena Pagnin, and Paul Stankovski Wagner. 2023. PAPR: Publicly Auditable Privacy Revocation for Anonymous Credentials. Cryptology ePrint Archive, Paper 2023/137. https://eprint.iacr.org/2023/137 https://eprint.iacr.org/2023/137.

[10] Joakim Brorsson and Martin Gunnarsson. 2023. Experiment results. https://anonymous.4open.science/r/DIPSAUCE_results-B155/.

[11] Joakim Brorsson and Martin Gunnarsson. 2023. Protocol and experiment code. https://anonymous.4open.science/r/contiki-ng-psa/README.md.

[12] Ignacio Cascudo and Bernardo David. 2020. ALBATROSS: publicly attestable batched randomness based on secret sharing. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. Springer, 311–341.

[13] Ryan Castellucci. 2013. libtprpg. https://github.com/ryancdotorg/libtprpg.

[14] T-H Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-preserving stream aggregation with fault tolerance. In *International Conference on Financial Cryptography and Data Security*. Springer, Berlin, Germany, 200–214.

[15] Melissa Chase and Sherman SM Chow. 2009. Improving privacy and security in multi-authority attribute-based encryption. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 121–130.

[16] Jing Chen and Silvio Micali. 2019. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science* 777 (2019), 155–183.

[17] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. 2018. Decentralized multi-client functional encryption for inner product. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, Berlin, Germany, 703–732.

[18] Jérémy Chotard, Edouard Dufour-Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. 2020. Dynamic decentralized functional encryption. In *Annual International Cryptology Conference*. Springer, Berlin, Germany, 747–775.

[19] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. 2021. GearBox: An Efficient UC Sharded Ledger Leveraging the Safety-Liveness Dichotomy. *IACR Cryptol. ePrint Arch.* 2021 (2021), 211.

[20] Keita Emura. 2017. Privacy-preserving aggregation of time-series data with public verifiability from simple assumptions. In *Australasian Conference on Information Security and Privacy*. Springer, Berlin, Germany, 193–213.

[21] Johannes Ernst and Alexander Koch. 2021. Private Stream Aggregation with Labels in the Standard Model. *Proc. Priv. Enhancing Technol.* 2021, 4 (2021), 117–138.

[22] Prosanta Gope and Biplab Sikdar. 2018. 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 (2018), 1554–1566.

[23] Trusted Computing Group. 2011. *TCG TPM Specification Version 1.2 - Part 1 Design Principles*. Technical Report. Trusted Computing Group, Beaverton, OR, United States.

[24] IEEE Computer Society. 2011. *IEEE Standard for Local and Metropolitan Area Networks, Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*.

[25] Texas Instruments. 2021. LAUNCHXL-CC1352R1 SimpleLink™ Multi-Band CC1352R Wireless MCU LaunchPad™ Development Kit. https://www.ti.com/tool/LAUNCHXL-CC1352R1. Accessed: 2022-01-21.

[26] Marc Joye and Benoît Libert. 2013. A scalable scheme for privacy-preserving aggregation of time-series data. In *International Conference on Financial Cryptography and Data Security*. Springer, Berlin, Germany, 111–125.

[27] Yasin Kabalci. 2016. A survey on smart metering and smart grid communication. *Renewable and Sustainable Energy Reviews* 57 (2016), 302–318.

[28] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. 2011. Privacy-friendly aggregation for the smart-grid. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, Berlin, Germany, 175–191.

[29] Ben Laurie. 2014. Certificate transparency. *Commun. ACM* 57, 10 (2014), 40–46.

[30] Iraklis Leontiadis, Kaoutar Elkhiyaoui, and Refik Molva. 2014. Private and dynamic time-series data aggregation with trust relaxation. In *International Conference on Cryptology and Network Security*. Springer, Berlin, Germany, 305–320.

[31] Lingjuan Lyu, Karthik Nandakumar, Ben Rubinstein, Jiong Jin, Justin Bedo, and Marimuthu Palaniswami. 2018. PPFA: Privacy preserving fog-enabled aggregation in smart grid. *IEEE Transactions on Industrial Informatics* 14, 8 (2018), 3733–3744.

[32] Stephanos Matsumoto and Raphael M Reischuk. 2017. IKP: turning a PKI around with decentralized automated incentives. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 410–426.

[33] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. {CONIKS}: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Berkeley, CA, United States, 383–398.

[34] Markus Meringer. 1999. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory* 30, 2 (1999), 137–146.

[35] Andrés Molina-Markham, Prashant Shenoy, Kevin Fu, Emmanuel Cecchet, and David Irwin. 2010. Private memoirs of a smart meter. In *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*. ACM, New York, NY, USA, 61–66.

[36] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. 2022. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* 18 (2022), 101089. https://doi.org/10.1016/j.softx.2022.101089

[37] Drand Organization. 2022. Drand - A Distributed Randomness Beacon Daemon. https://github.com/drand/drand.

[38] Rafail Ostrovsky and Moti Yung. 1991. How to withstand mobile virus attacks. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 51–59.

[39] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[40] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252. https://doi.org/10.17487/RFC7252
[41] Elaine Shi, T-H. Hubert Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-Preserving Aggregation of Time-Series Data. *Network and Distributed System Security Symposium, NDSS* 1 (2011), 17.
[42] János Szigeti. 2022. Big Unsigned Integers. https://github.com/SzigetiJ/biguint.
[43] Jonathan Takeshita, Zachariah Carmichael, Ryan Karl, and Taeho Jung. 2023. TERSE: Tiny Encryptions and Really Speedy Execution for Post-Quantum Private Stream Aggregation. In *Security and Privacy in Communication Networks*, Fengjun Li, Kaitai Liang, Zhiqiang Lin, and Sokratis K. Katsikas (Eds.). Springer Nature Switzerland, Cham, 331–352.
[44] Jonathan Takeshita, Ryan Karl, Ting Gong, and Taeho Jung. 2020. SLAP: Simple Lattice-Based Private Stream Aggregation Protocol. Cryptology ePrint Archive, Paper 2020/1611. https://eprint.iacr.org/2020/1611 https://eprint.iacr.org/2020/1611.
[45] Hendrik Waldner, Tilen Marc, Miha Stopar, and Michel Abdalla. 2021. Private Stream Aggregation from Labeled Secret Sharing Schemes. Cryptology ePrint Archive, Paper 2021/081. https://eprint.iacr.org/2021/081 https://eprint.iacr.org/2021/081.

A EXPERIMENTAL SETUP

A.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, ECC, 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.

A.2 Operating System and Software

The experiments are implemented on the Contiki-NG operating system [36], 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 [42] to perform 128-bit arithmetics, and the libtprpg [13] library to generate the pseudo-random permutation used in DIPSAUCE.

A.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.

A.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 7 illustrates the setup.

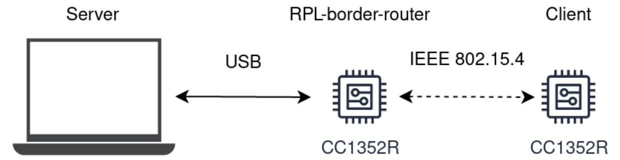


Figure 7: An illustration of the experimental setup.

.

B PYTHON CODE FOR CALCULATING ADVANTAGE FOR DIFFERENT n, t, k

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

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

    # 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)
```