Differential Privacy by Typing in Security Protocols

Fabienne Eigner  Matteo Maffei
Saarland University, Germany
{eigner,maffei}@cs.uni-saarland.de

Abstract—Differential privacy is a confidentiality property for database queries which allows for the release of statistical information about the content of a database without disclosing personal data. The variety of database queries and enforcement mechanisms has recently sparked the development of a number of mechanized proof techniques for differential privacy.

Personal data, however, are often spread across multiple databases and queries have to be jointly computed by multiple, possibly malicious, parties. Many cryptographic protocols have been proposed to protect the data in transit on the network and to achieve differential privacy in a distributed, adversarial setting. Proving differential privacy for such protocols is hard and, unfortunately, out of the scope of the aforementioned mechanized proof techniques.

In this work, we present the first framework for the mechanized verification of distributed differential privacy. We propose a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders and can be used to reason about compromised parties. Furthermore, we develop a linear, distance-aware type system to statically and automatically enforce distributed differential privacy in cryptographic protocol implementations (expressed in the RCF calculus). We also provide an algorithmic variant of our type system, which we prove sound and complete. Finally, we tested our analysis technique on a recently proposed protocol for privacy-preserving web analytics: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

I. INTRODUCTION

Personal information (e.g., patient records, browsing histories, social graphs, and behavioral data used for advertising) is today disseminated in a wealth of databases spread across different institutions and services. On the one hand, disclosing information about these data is often desirable for improving services, analyzing trends, performing marketing studies, conducting research, and so on. On the other hand, this information leakage may irremediably compromise the privacy of users. Narayanan and Shmatikov [1] have shown that anonymized and aggregated data, which at first glance look seemingly harmless, may actually reveal an incredible amount of information on each user. The research community has long struggled to understand what privacy means in the context of database queries and how to measure the amount of information leaked by each query.

Differential privacy. Today, differential privacy [2] is recognized as one of the fundamental notions of privacy for queries on statistical databases. Intuitively, a query is differentially private if it behaves statistically similarly on any pair of databases differing in one entry. In other words, the contribution of each single entry to the query result is bounded by a small constant factor, even if all remaining entries are known. A deterministic query can be made differentially private by perturbing the result with a certain amount of noise, thus reducing the accuracy of the answer.

Mechanized certification of differential privacy. Many works on differential privacy focus on specific classes of queries and noise mechanisms, proving properties thereof manually and in an ad-hoc manner. The disadvantage of this approach is that each new database type or each new query requires its own separate proof. Taking up this challenge, recent research focused on the development of mechanized certification techniques for differential privacy. For instance, Reed and Pierce [3] showed how to automatically and statically enforce differential privacy for a large class of database queries based on a type system for a higher-order functional language. Gaboardi et al. [4] extended this type system to allow the certification of a larger class of queries whose sensitivity might depend on runtime information. Barthe et al. have presented CertiPriv [5], a mechanized framework based on interactive theorem proving that can be used to derive formal guarantees of differential privacy for a variety of sanitization mechanisms. For a more detailed discussion of the related work, we refer to Section IX.

Distributed differential privacy. So far, we focused on the concept of differential privacy for single databases. In reality, data are often distributed across different databases, or stored on personal devices, and one has to compute statistical functions on the dataset obtained by joining these data, yet in a privacy-preserving manner. Since data may be read and manipulated by network attackers as well as compromised parties, these computations necessarily involve cryptographic protocols. A growing body of recent work has been focusing on the development of cryptographic protocols for distributed differential privacy (e.g., verifiable secret sharing [6], secure function evaluations [7], secure multi-party computations [8], multi-party distributed data aggregation algorithms [9]–[13], and local learning algorithms [14]) and on computational definitions of differential privacy against polynomially-bounded opponents [8]. So far, however, the differential privacy guarantees offered by such protocols had to be proven by hand.

Our contributions. We introduce the first mechanized
verification technique for distributed differential privacy, taking the first steps to reconcile the formal analysis of cryptographic protocols with the growing body of work on the formal certification of differential privacy guarantees. Specifically, we propose a symbolic definition of differential privacy for distributed databases. Our definition considers an attacker model that takes Dolev-Yao intruders\(^1\) into account and can also be used to reason about compromised parties. Furthermore, we present a linear type system to statically enforce this privacy property in cryptographic protocol implementations. Our framework uniformly captures a variety of perturbation mechanisms, such as Laplace noise addition [15] and the exponential mechanism by McSherry and Talwar [16].

We also provide a sound and complete algorithmic variant of our type system, which allows for automating the analysis. Finally, we tested our analysis technique on a protocol for privacy-preserving web analytics [17]: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

**Outline.** The paper is organized as follows: Section II introduces the symbolic definition of distributed differential privacy, Section III presents the calculus, Section IV explains the link between differential privacy and typing, Section V illustrates the type system, Section VI gives some insights into the algorithmic variant of the type system, Section VII discusses our symbolic model of cryptography, Section VIII shows our type system at work on a cryptographic protocol for non-tracking web analytics, Section IX discusses the related work, and Section X concludes and gives directions of future research. In Appendix A we exemplify how to extend our type system to other sanitization mechanisms, e.g., the exponential one [16]. For proofs, technical details of the (algorithmic) type system, and the code of both the cryptographic library and the case study we refer to the long version [18].

II. DISTRIBUTED DIFFERENTIAL PRIVACY

In this section we explain the key ideas behind our definition of distributed differential privacy. We also demonstrate the need for a mechanized proof technique for distributed differential privacy by showing a previously overlooked attack against a protocol on distributed databases.

**Differential privacy.** A query is differentially private if it behaves statistically similarly on all databases \(D, D'\) differing in one entry, written \(D \sim D'\). This means that the result of the query is not significantly changed by the presence or absence of each individual database entry.

The definition of differential privacy is parameterized by a number \(\epsilon\), which *measures* how strong the privacy guarantee is: the higher \(\epsilon\), the stronger the risk to join the database.

---

\(^1\)A Dolev-Yao intruder can overhear, intercept, and synthesize the cryptographic messages exchanged on the network, but it cannot break the cryptographic algorithms.

**Definition 1** (Differential Privacy [2]). A randomized function \(f\) is \(\epsilon\)-differentially private iff for all databases \(D, D'\) such that \(D \sim D'\) and every set \(S \subseteq \text{Range}(f)\),

\[
Pr[f(D) \in S] \leq e^\epsilon \cdot Pr[f(D') \in S]
\]

A deterministic query can be made \(\epsilon\)-differentially private by perturbing the result with a certain amount of noise, thus reducing the accuracy of the answer. We will describe some of these perturbation mechanisms throughout the paper.

**Distributed differential privacy.** As previously mentioned, data are frequently distributed across different databases or stored on user devices, and it is often desirable to compute statistical functions on the dataset obtained by joining these data. The exchange of data such as query results or fragments of a local database over an untrusted network necessarily involves cryptographic protocols.

Distributed differential privacy has to be defined with respect to the overall protocol and, particularly, against an opponent that can query the databases as well as interfere with the cryptographic messages exchanged on the network (e.g., by forging messages or replaying them).

The intuition underlying our definition of differential privacy is to think of the protocol as a query mechanism at disposal of the opponent. The database is distributed among protocol participants and, by interacting with them, the opponent can learn information about the database. The opponent is active in that it can choose how to interact with protocol participants based on the information acquired in the previous message exchanges. As usual in protocol analysis [19], we formalize the protocol as a function, which takes as input the database and can be called by the opponent. Typically, this function creates the cryptographic material, distributes the database across the protocol participants, and finally returns the functions implementing each of these parties. The opponent can call and schedule the execution of these functions at will.

Intuitively, a protocol \(P\) is \(\epsilon\)-differentially private if for all databases \(D, D'\) differing in a single entry, the probability that the opponent outputs 1 when interacting with \(PD\) is approximately the same as when interacting with \(PD'\).

**Definition 2** (Distributed differential privacy (\(\epsilon\)-DDP)). \(P\) is \(\epsilon\)-differentially private iff for all databases \(D, D'\) such that \(D \sim D'\) and all opponents \(O\),

\[
Pr[O(PD) \rightarrow^* 1] \leq e^\epsilon \cdot Pr[O(PD') \rightarrow^* 1]
\]

This definition is the symbolic counterpart of the definition of \(\epsilon_k\)-IND-CDP differential privacy for interactive protocols against polynomially-bounded opponents proposed by Mironov et al. [8]. Here the attacker is not polynomially bounded, since we work in a symbolic setting and under the perfect cryptography assumption: the Dolev-Yao attacker can only access the cryptographic libraries exported by the
program, which model the idealized semantics of cryptographic primitives by constructs of the language and are, thus, suitable for automated verification.

Furthermore, we remark that the above definition can be used to reason about malicious parties, a common ingredient in the threat model of cryptographic protocols for distributed differential privacy, by simply letting these parties be under the control of the attacker (cf. Section VIII).

Finally, notice that our definition can be used to reason about secrecy properties of cryptographic protocols in general, by simply letting the database be the set of secrets of protocol participants and by splitting the database so as to give each participant her own secret. In contrast to the existing symbolic definitions of secrecy for cryptographic protocols, which are based on reachability properties [20] or observational equivalence relations [21], this definition is quantitative in that it provides a bound on the amount of sensitive information leaked in a certain protocol execution.

What can go wrong? Many new systems for distributed differential privacy are emerging. Analyzing the privacy properties of these systems by hand is not only tedious but, due to the complex nature of the systems and their underlying cryptographic protocols, also prone to overlooking attacks. For instance, we discovered a previously unknown attack on a recently proposed protocol for privacy-preserving web analytics by Akkus et al. [17].

The core part of this protocol is depicted in Table I. Intuitively, the provider of a website (publisher) uses a third party web analytics service (data aggregator) to gain aggregated information about the users visiting the site (clients). Such information can include user demographics (e.g., “how many men over 50 visited the website last week?”), browsing behavior, and information about the clients’ systems. In order to achieve this goal in a privacy-preserving manner, the publisher acts as a proxy between the clients and the aggregator: it collects encrypted information sent by multiple clients visiting the website, mixes them with some fake “noisy” data, and forwards them to the aggregator. The aggregator decrypts all received ciphertexts and (not being able to distinguish between real and fake user information) combines the results. It adds noise to the produced analytics and forwards the “double noisy” results to the publisher.

Overall, both the publisher and the data aggregator obtain analytics about the clients that visited the publisher’s webpage, but both results are not exact: the publisher cannot remove the noise that the aggregator added and vice-versa. Assuming that the noise mechanisms used by the publisher and aggregator to perturb the client data are differentially private and that the publisher and the data aggregator do not collude, one would assume that the overall protocol can be proved to enforce distributed differential privacy. As it turns out, this is not the case.

Consider the following new “snapshot” attack. Upon receiving the encrypted data of an individual client, a malicious publisher can duplicate these values multiple times and forward several copies either directly to the data aggregator, or to another publisher, thus allowing this client’s data to influence the overall aggregated result multiple times. This means that, even if a client stops answering queries about its personal data after its privacy budget is spent, the publisher still possesses a “snapshot” of the previous answers and can replay them. The privacy budget is thus not under the control of the client, making it impossible to show \( \epsilon \)-DDP for a fixed \( \epsilon \).

In the remainder of this paper, we introduce a type system to statically enforce differential privacy in cryptographic protocol implementations. We use our type system to analyze the previously illustrated protocol in Section VIII, where we give more details about the attack, propose a fix, and verify the resulting variant.

### III. Calculus

The \( \lambda \)-calculus used in this paper is a dialect of RCF, a formal core of F# introduced by Bengtson et al. [19] to reason about cryptographic protocol implementations.

\(^2\) **privacy budget**: the number of queries a client can safely answer in order to still guarantee \( \epsilon \)-DDP.
\[
\begin{align*}
\alpha &::= \text{inl} \mid \text{inr} \mid \text{fold} \\
h &::= x \mid c \mid f \\
M, N, D &::= x \\
c &::= \lambda x.A \\
f &::= (M, N) \\
h &::= M \\
\lambda x.A &::= \text{read}_{a, r} \\
\text{write}_{a, r} &::= A, B, P, Q ::= M \\
M &::= \text{let } x = A \text{ in } B \\
&::= \text{case } M \text{ of } x \text{ in } A \text{ else } B \\
&::= \text{unfold } M \text{ as } x \text{ in } A \text{ else } B \\
&::= \text{ref}_x \\
&::= \text{add}_\text{noise}_a M
\end{align*}
\]

**Syntax.** The syntax of the calculus (cf. Table II) includes standard functional constructors, values, and expressions. The language further supports Laplace noise addition and references, by providing reference creators, readers, and writers. References are pairs of functions that read and write on a memory location. The execution of \(\text{ref}_x\) allocates a fresh label \(a\) and returns a pair \((\text{read}_{a, r}, \text{write}_{a, r})\) of functions for reading and writing at location \(a\). We statically annotate references with the type \(\tau\) of the values stored therein. Note that type annotations do not have any semantic import.

The calculus and the type system are parameterized by a signature, which exports constants, functions, and the respective types, as discussed in Section IV. This enhances the expressivity of our framework by making it extensible to new primitives. We assume that the constants exported by the signature include reals, ranged over by \(r\), the unit value \(\bot\), and sets, ranged over by \(S\). We can encode boolean values as true \(\triangleq \text{inl} \bot\) and false \(\triangleq \text{inr} \bot\). The constructors some and none for option types can be encoded similarly.

**Remark (Modeling Cryptographic Protocols).** In order to model cryptographic protocols, we follow the approach proposed by Fournet et al. [22], for which computational soundness guarantees have been proved: the protocol is modeled as a function that is given to the opponent, who acts as a scheduler. In our setting such a protocol function expects a secret database as an input, splits the database entries amongst the \(n\) honest protocol participants, and returns \(n\) functions that model the behavior of each individual participant. If a participant is required to perform multiple message exchanges, the corresponding function is cascaded, expecting a message from the network as an input, and returning both the outgoing message and the code of the continuation function, which models the remaining protocol steps of that participant.

**Semantics.** We formalize the semantics of the calculus (cf. Table III) using a labelled probabilistic reduction relation \([S, A] \xrightarrow{\ell} [S', A']\) between configurations \([S, A]\) consisting of a store \(S\) and an expression \(A\). We track the probability \(p\) that a certain reduction takes place as well as the rule \(\ell\) applied. The only non-deterministic primitive is \(\text{add}_\text{noise}_a r_1\), which returns \(r_1 + r_2\), where \(r_2\) is drawn according to the Laplace distribution \(\text{Lap}_p\) that has the density function \(Pr[x] = \frac{1}{2p}e^{-|x|/p}\). Following common praxis in the literature on differential privacy [15], we use \(Pr[x]\) to denote probability density for both continuous and discrete random variables. In Section IV we show that Laplace noise addition is crucial to achieve \(\epsilon\)-differential privacy. The label \(\text{Noise}(r_1, r_2, s)\) keeps track of the arguments \(r_1, r_2\) and of the parameter \(s\) of the Laplace distribution.

**Remark (Discrete Domains).** We tacitly assume all types and the constants exported by the signature (including reals) to range over discrete domains. For example, we can consider the set of reals in fixed point notation, i.e.,
\[ \mathbb{R}_{\text{disc}} = \{z/a \mid z \in \mathbb{Z}\} \] for some (arbitrarily large) positive natural number \(a\) (usually a power of 2 or 10). For convenience, we usually just write \(R\) instead of \(\mathbb{R}_{\text{disc}}\). Let \(x = z/a\): formally, we use \(\operatorname{Lap}_s(x)\) to denote \(Pr[z/a \leq X < (z+1)/a]\) for the continuous random variable \(X\) drawn according to the Laplace distribution, i.e., we integrate the Laplace density function over the interval \([z/a, (z+1)/a]\).

In practice, approximating continuous domains by discrete ones is necessary to represent them in computers.

**Remark (Probabilistic Semantics).** Reed and Pierce [3] model perturbed results as distributions that are represented by monads and are given a deterministic, big-step denotational semantics. We found it more convenient to work with a probabilistic, small-step operational semantics, which is closer to the semantics traditionally used in type systems for cryptographic protocols and allows for leveraging existing proof techniques.

The remaining deterministic primitives are labelled with \(\text{det}\). The store is a finite map from references to values. The content of each reference can be either none, if the reference is empty, or some \(M\). Departing from RCF, we define the semantics of references in a message-passing style, which is reminiscent of the concept of M-structures [23]: data are automatically removed from the referenced memory after being read, thus preventing data duplication. This choice simplifies the formalization of our linear type system and does not affect the expressivity of the language, since destructive and non-destructive read operators can be obtained from each other by encoding (e.g., non-destructive reads can be encoded by rewriting the read data).

Finally, the semantics of the calculus is parameterized by the semantics of the functions exported by \(\Sigma\). These functions must be deterministic, take as argument functional terms, and return constant terms, which are defined below:

\[
F \quad ::= \quad f \mid c \mid (F_1, F_2) \mid h F \quad \text{functional term} \\
C \quad ::= \quad c \mid (C_1, C_2) \mid h C \quad \text{constant term}
\]

Notice that these functions may be higher-order and can be called by passing the arguments in uncurried form. Enforcing a clear separation between the functions exported by the signature and the values of our calculus is crucial to leverage existing results on function sensitivity and, thus, to make our framework easily extensible to new primitives and types. We write \(f(F) =_\Sigma C\) to denote that \(C\) is the result of \(f(F)\) in \(\Sigma\) and just use \(f(F)\) to denote \(C\) whenever \(\Sigma\) is clear from the context.

We finally define the probability that a certain expression \(P\) reduces into another expression \(Q\), as required by the definition of \(\epsilon\)-differential privacy stated in Section I.

**Definition 3 (Reduction probability).** For all expressions \(P, Q\) and probabilities \(p, r\), all \(n \in \mathbb{N}^+\), and all evaluation rules \(\ell_i\) for \(i \in [1, n]\),

\[
\tau, \rho ::= 1_k \phi \quad \text{type} \quad (k \in \mathbb{R}_{\geq 0} \cup \{\infty\}) \\
\phi, \psi ::= \quad b \quad \text{base type} \\
\alpha \quad \text{type variable} \\
\mu, \nu \quad \text{iso-recursive type} \quad (\alpha \text{ bound in } \tau) \\
\tau + \tau \quad \text{sum type} \\
\tau \otimes \tau \quad \text{multiplicative pair type} \\
\tau \rightarrow \tau \quad \text{function type}
\]

**Convention:** We write \(!_{k}(1_k \phi)\) to denote \(1_k \cdot \psi \cdot \phi\).

**Table IV**

<table>
<thead>
<tr>
<th>Syntax of Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P \xrightarrow{\ell_1, \ldots, \ell_n} Q) if there exist (S, S', P') such that ([0, P] \xrightarrow{\ell_1, \ldots, \ell_n} [S', P']) and ([S', P'] \xrightarrow{\ell_1, \ldots, \ell_n} [S, Q]).</td>
</tr>
<tr>
<td>(Pr[P \xrightarrow{\tau} Q] = \sum_{\ell_1, \ldots, \ell_n} P \xrightarrow{\ell_1, \ldots, \ell_n} \sum_{\tau} S, Q p).</td>
</tr>
</tbody>
</table>

**IV. Differential Privacy by Typing**

In this section we explain the intrinsic interplay between differential privacy and types in more detail.

As previously mentioned, a query can be made differentially private by perturbing the result with some noise. An important observation is that the amount of noise depends on the query: the more a single entry contributes to the query result, the stronger the noise has to be. The property we are interested in is the sensitivity of queries to quantitative differences in their inputs [15]. The sensitivity of a function measures how much this function amplifies the distance between inputs. Intuitively, queries of low sensitivity map nearby inputs to nearby outputs. The distance between values depends on their type: for the query below, the distance between databases is the Hamming distance, while the distance between real numbers is the Euclidean one. For instance, the query "how many foreigners are registered in the hospital database?" has sensitivity 1, since adding or removing a single entry will change the result by at most 1.

Reed and Pierce [3] showed there is an intimate connection between resource usage and function sensitivity: for instance, a deterministic function \(f\) that uses each entry of the database only \(k\) times and calls only \(1\)-sensitive functions is at most \(k\)-sensitive, that is, it can magnify the distance of the inputs at most by a factor of \(k\). Based on this intuition, they proposed a linear type system that statically bounds the usage of resources and, thus, can be used to statically over-approximate the sensitivity of a function and to enforce differential privacy.

**Types.** Table IV shows the syntax of types. As usual in linear type systems, the type of a value serves a double purpose: it describes the nature of the value (e.g., real number) as well as the number of times this value can be used at runtime. Consequently, the syntax of types is defined by mutual induction around the concept of type and core type.

Types of the form \(1_k \phi\), with replication index \(k\), describe values of core type \(\phi\) that can be used at most \(k\) times at run-
time. In this sense, our type system is affine, and thus more liberal than a linear type system, since it enforces an upper bound on the number of times a certain resource is used, as opposed to the exact number. In particular, if a function is given type \( !_k \phi \rightarrow \tau \), then the argument can be used at most \( k \) times in the body of the function. Notice that values of type \( !_k \phi \) can be used arbitrarily often and we call these types exponential. For the sake of readability, we often omit replication index \( ! \). Replication indexes are non-negative real numbers (i.e., \( k \in \mathbb{R}^+ \cup \{\infty\} \)). Here a replication index of \( 0 \) denotes that a value cannot be used, while replication indexes \( k \) such that \( 0 < k < 1 \) are helpful to express a sensitivity < 1. Core types comprise the base types \( b \) defined in the signature \( \Sigma \), type variables, iso-recursive types, sum types, pair types, and function types. Note that the system by Reed and Pierce [3] includes additive pair types of the form \( \tau \& \tau \); although we could easily add them to our system, we decided to omit them for simplicity, since they are not useful for the examples we considered.

**Distance on types.** We require a notion of distance for all types and core types in our type system. Note that this means that the signature has to provide a metric for all base types. We adopt the metric for types introduced by Reed and Pierce [3], which we overview below. For types with replication index \( k \), we define the distance as the distance of the core type multiplied by \( k \), thus \( \delta_{\cdot \cdot}(x, y) = k \cdot \delta_{\cdot}(x, y) \). For core pair types, the distance is defined as the sum of the distances of the components \( \delta_{\cdot \cdot}(x_1, y_1), (y_1, y_2) = \delta_{\cdot}(x_1, x_2) + \delta_{\cdot}(y_1, y_2) \). The distance of functions is defined as the maximal distance of the outputs that the two functions produce for the same input \( \delta_{\cdot \cdot}(f, g) = \max_{x \in \tau} (\delta_{\cdot}(f(x), g(x))) \). The distance on core sum types is defined as

\[
\delta_{\tau + \rho}(x, y) = \begin{cases} 
\delta_{\cdot}(x', y') & \text{if } x = \text{inl } x' \text{ and } y = \text{inl } y' \\
\delta_{\cdot}(x', y') & \text{if } x = \text{inr } x' \text{ and } y = \text{inr } y' \\
\infty & \text{otherwise}
\end{cases}
\]

Intuitively, \( \text{inl } x \) and \( \text{inr } x \) have distance \( \infty \) since one can perform a pattern-matching operation on them and, based on the result, produce arbitrarily distant results. The distance of two values of an iso-recursive core type is intuitively defined by unfolding the two values. Note that the following definition \( \delta_{\mu \alpha \cdot \tau}(\text{fold } x, \text{fold } y) = \delta_{\cdot \cdot}(\mu \alpha \cdot \tau)(x, y) \) of distance for iso-recursive types is not well-founded in all cases (e.g., for the type \( \mu \alpha \cdot \alpha \)). We do not consider types for which this distance is not well-founded, as they are not needed in any practical example we considered.

**Example 1.** \( \delta_{\cdot \cdot}(\mathbb{R} \oplus \mathbb{R} \oplus \mathbb{R})(1, 2, (0, 3)) = 3 \cdot (1 + 1) = 6 \) and \( \delta_{\mathbb{R} \rightarrow \mathbb{R}}(\lambda x. x, \lambda x. x + 1) = 1 \) and \( \delta_{\mathbb{R} \rightarrow \mathbb{R}}(\lambda x. x, \lambda x. x + x) = \infty \).

**Signature.** As previously mentioned, our framework is parameterized by a signature \( \Sigma \), which exports constants, functions, and the respective types. In this work, we assume that real numbers are given type \( \mathbb{R} \), the unit value type \( \text{Unit} \), while sets of values of type \( \tau \) are given type \( \text{Set}(\tau) \). The signature additionally needs to provide a distance \( \delta_b \) for each base type \( b \). \( \delta_b \) is the Euclidean distance between numbers, \( \delta_{\text{Unit}} \) is the null distance, while \( \delta_{\text{Set}(\tau)} \) is the symmetric difference (the number of entries that are contained in one but not in the other set). Given these types, we can define standard encodings for the (core) types \( \text{Bool}, \text{Option}(\tau), \) and \( \text{List}(\tau) \), as shown below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Bool}</td>
<td>( \Delta_{\text{Unit} + \text{Unit}} )</td>
<td>boolean type</td>
</tr>
<tr>
<td>\text{Option}(\tau)</td>
<td>( \Delta_{\text{Unit} + \tau} )</td>
<td>option type</td>
</tr>
<tr>
<td>\text{List}(\tau)</td>
<td>( \Delta_{\mu \alpha. \text{Unit} + (\text{Unit} \otimes \alpha)} )</td>
<td>list type</td>
</tr>
</tbody>
</table>

**Type-based \( k \)-sensitivity and differential privacy.** The notion of sensitivity can be generalized to arbitrary types \( \tau \) for which a metric \( \delta_\tau \) exists, as shown below:

**Definition 4** (Type-Based \( k \)-Sensitivity [3]). A function \( f \) is \( k \)-sensitive in \( \tau_1 \rightarrow \tau_2 \) iff \( \delta_{\tau_2}(f(x), f(y)) \leq k \cdot \delta_{\tau_1}(x, y) \) for all \( x, y \in \tau_1 \).

**Example 2.** The function \( f_1(x, y) = x + y \) is 1-sensitive in \((\mathbb{R} \oplus \mathbb{R}) \rightarrow \mathbb{R} \). Notice that each argument is used only once. The function \( f_2(x) = 3x \) has sensitivity 3 in \( \mathbb{R} \rightarrow \mathbb{R} \). One might say that here the argument is used only twice, but since multiplication is defined using addition, \( f_2(x) = x + x + x \) in fact uses the argument three times.

The sensitivity of a function determines the amount of noise that one has to add to the result for rendering the function in question \( \epsilon \)-differentially private.

**Proposition 1** (Sensitivity and DP [15]). Suppose \( f \) is \( k \)-sensitive in \( \mathcal{D}^n \rightarrow \mathbb{R} \). Define the random function \( q = \lambda x. \text{add\_noise}_{k/\epsilon}(x) \). Then \( q \) is \( \epsilon \)-differentially private.

Notice that \( k \)-sensitivity can be reduced to 1-sensitivity.

**Proposition 2** (\( k \)-Sensitivity vs 1-Sensitivity [3]). A function \( f \) is \( k \)-sensitive in \( \tau_1 \rightarrow \tau_2 \) if and only if it is a 1-sensitive function in \( \lambda_k \tau_1 \rightarrow \tau_2 \).

**Example 3.** The function \( f_2 \) defined in Example 2 is 1-sensitive in type \( \lambda_k \mathbb{R} \rightarrow \mathbb{R} \).

In the type system by Reed and Pierce, \( \lambda_k \tau_1 \rightarrow \tau_2 \) is the type of \( k \)-sensitive functions in \( \tau_1 \rightarrow \tau_2 \) (or, equivalently, 1-sensitive functions in \( \lambda_k \tau_1 \rightarrow \tau_2 \)). This holds true for deterministic functions in our type system and, in the next section, we generalize this property to stateful randomized expressions.

V. **OUR TYPE SYSTEM**

In this section, we introduce a distance-aware type system that enforces \( \epsilon \)-differential privacy in cryptographic protocol implementations. The key idea is to enforce a distance preservation property for well-typed configurations, which is close in spirit to the notion of sensitivity for deterministic
functions but additionally takes into account the store and the reduction probabilities. Given two configurations that share the same structure and only differ in some of their constants, their distance is defined by summing the distance of the differing constants. Intuitively, the distance preservation property says that two well-typed configurations sharing the same structure reduce with approximately the same probability into configurations whose distance does not exceed the distance of the initial configurations.

**Typing environment and judgments.** The typing environment is a list of type bindings for variables, kind bindings for type variables, and type variable declarations of the form $x : \tau, \alpha : \kappa$, and $\alpha$, respectively. The type system comprises six typing judgments: the well-formedness judgments $\Gamma \vdash \emptyset$ and $\Gamma \vdash \tau$ for typing environments and types, respectively; the kinding judgments $\Gamma, \alpha : \kappa \vdash \kappa$ and $\Gamma \vdash \phi :: \kappa$, which classify types and core types based on whether the corresponding values may be sent to the opponent (kind pub) or be received from the opponent (kind tnt); the subtyping judgment $\Gamma \vdash \tau <: \rho$, which enhances the expressivity of the type system by allowing values of type $\tau$ to be used in place of values of type $\rho$; and the typing judgment for expressions $\Gamma \vdash A : \tau$.

**Well-formedness judgments.** The well-formedness rules are stated in Table V. A type $\tau$ is well-formed in $\Gamma$ if the type variables occurring free in $\tau$ are bound in $\Gamma$. A typing environment $\Gamma$ is well-formed if the variables bound in $\Gamma$ are all distinct and their types are well-formed.

The domain of environments is defined as $\text{dom}() \equiv \emptyset$, $\text{dom}(\Gamma, \alpha) \equiv \text{dom}(\Gamma) \cup \{\alpha\}$, $\text{dom}(\Gamma, \alpha :: \kappa) \equiv \text{dom}(\Gamma) \cup \{\alpha\}$, and $\text{dom}(\Gamma, x : \tau) \equiv \text{dom}(\Gamma) \cup \{x\}$. The set of free type variables in $\tau = !k\phi$ is denoted by $ft(\tau) \equiv ft(\phi)$, where $ft(x : \tau) \equiv ft(\tau)$ and $ft(\alpha) \equiv \emptyset$.

**Kinding and subtyping.** One of the standard ingredients of type systems for cryptographic protocols is the kinding relation [19, 24]: a type has kind public if messages of that type can be sent to the opponent and kind tainted if messages of that type can be received from the opponent. The types that are both public and tainted are equivalent by subtyping (i.e., they are subtypes of each other). This is crucial to prove the opponent typability lemma which says that all opponents are well-typed. This property allows us to assume that the attacker is well-typed in the proof of soundness without fearing that our typing discipline limits the power of the attacker. A property of our type system is that a type is both public and tainted if and only if all replication indexes occurring therein are set to $\infty$. We call such types opponent types. Formally, we write $[\tau]_{\infty}$ to denote the type obtained by replacing all replication indexes in $\tau$ with $\infty$ and we define the set $\mathcal{OPP}$ of opponent types as the set of types $\tau$ such that $\tau = [\tau]_{\infty}$.

The kinding and subtyping relations are reported in Table VI. Since the rules are mostly standard [19], we only comment the interesting ones. A type is public only if it has the form $!k\phi$, since the opponent can use the received values arbitrarily often, and $\phi$ is public (Kind PUB). A type of the form $!k\phi$ is tainted only if $\phi$ is tainted (Kind Tnt): we do not place any constraint on $k$, since we can choose to limit the number of times our code is allowed to use the values received from the opponent. The core types defined by $\Sigma$ are both public and tainted, since the constants in the signature can be used by the opponent (Kind Sig). With regards to subtyping, the only aspect that is worth pointing out is that the subtyping relation is contravariant in the replication index, i.e., $\Gamma \vdash !k\tau <!j\tau$ only if $j \leq k$. 

---

**Table V**

<table>
<thead>
<tr>
<th>Type</th>
<th>$\Gamma \vdash \emptyset$ ft($\theta$) $\subseteq \text{dom}(\Gamma)$</th>
<th>Env Empty</th>
<th>$\emptyset \vdash \emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env Entry</td>
<td>$\Gamma \vdash \emptyset$ ft($\mu$) $\subseteq \text{dom}(\Gamma)$ dom($\mu$) $\cap \text{dom}(\Gamma) = \emptyset$</td>
<td>$\Gamma, \mu \vdash \emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

**Notation:** We write $\theta$ to denote both types and core types.

---

<table>
<thead>
<tr>
<th>Type</th>
<th>$\Gamma \vdash \tau : \kappa$</th>
<th>Env Empty</th>
<th>$\emptyset \vdash \tau : \kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env Entry</td>
<td>$\Gamma \vdash \tau : \kappa$ ft($\mu$) $\subseteq \text{dom}(\Gamma)$ dom($\mu$) $\cap \text{dom}(\Gamma) = \emptyset$</td>
<td>$\Gamma, \mu \vdash \tau : \kappa$</td>
<td></td>
</tr>
</tbody>
</table>

---

**Table VI**

<table>
<thead>
<tr>
<th>Sub Kind</th>
<th>$\Gamma \vdash \tau : \alpha$ $\vdash \rho : \kappa$</th>
<th>Sub Repl</th>
<th>$\Gamma \vdash \phi :: !k\psi$ $k \leq t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub Repl</td>
<td>$\Gamma \vdash \tau : \alpha$ $\vdash \rho : \kappa$</td>
<td>Sub Sum</td>
<td>$\Gamma \vdash \tau : \alpha$ $\vdash \rho : \kappa$</td>
</tr>
<tr>
<td>Sub Pair</td>
<td>$\Gamma \vdash \tau : \alpha$ $\vdash \rho : \kappa$</td>
<td>Sub Fun</td>
<td>$\Gamma \vdash \tau : \alpha$ $\vdash \rho : \kappa$</td>
</tr>
</tbody>
</table>

**Notation:** pub = tnt and tnt = pub.
In particular, this means that values of type $!_{k}\tau$ for $k < \infty$ can never be sent to the opponent, since public types have replication index $\infty$ and the replication index can never be increased by subtyping. This is crucial for the soundness of the type system, since the opponent can use any value it receives arbitrarily often, in particular $k+1$ times. However, values received from the attacker at type $!_{\infty}\tau$ can be super-typed to $!\tau$ and be treated as confidential.

**Typing values and expressions.** One of the goals of our type system is to statically enforce that values of type $!_{k}\tau$ are used at most $k$ times at run-time. Intuitively, this is achieved by formalizing the typing derivations for values and expressions in a way that the environment used in the thesis is the sum of the environments used in the hypotheses (or reading the typing rules upside down, the environment is split along the typing derivations). This prevents multiple usages of the same resource.

The sum of environments is formalized in Table VII. Type variables and kind bindings are not subject to cardinality constraints, since they are not associated to any values.\(^3\) If the two typing environments contain type bindings of the form $x : !_{k}\tau$ and $x : !_{k'}\tau$, respectively, their sum returns an environment containing $x : !_{k+k'}\tau$.

**Values.** The rules for typing values are reported in Table VIII and are mostly standard [3]. For typing pairs, one has to type each of the components individually and sum the typing environments, thus avoiding duplication of resources ($\ominus I$). As discussed in Section IV, the signature $\Sigma$ defines a core type $\phi$ for each constant and function. These are given the core type specified in $\Sigma$ with a linear replication factor ($\Sigma I$). The function $\text{read}_{a:\tau}$ to read from reference $a$ is given type $!_{1}(\text{Unit} \rightarrow !_{1}\text{Option}(\tau))$ (READ), since it takes as input the unit value and returns either some $M$, if $M$ is the value currently stored in the reference, or none, if the reference is empty. Rule (READ OPP) applies to references whose content is not subject to any cardinality constraint (i.e., $\tau \in \text{OPP}$); in this case, the value read from the reference can be used arbitrarily often and the replication index of the option type can thus be $\infty$. This rule is needed to type-check the opponent. The function $\text{write}_{a:\tau}$ is instead given type $!_{1}(\tau \rightarrow !_{\infty}\text{Unit})$, since it takes as input a value of type $\tau$ to be stored in the reference and returns the unit value. Finally, we can introduce types with arbitrary positive replication index using rule $\mu\eta!$: if $M$ is given type $!_{1}\phi$ under $\Gamma$, then it is possible to type-check $M$ with type $!_{k}\phi$ given the environment $k\Gamma$.

**Expressions.** The typing rules for expressions are reported in Table IX. The subsumption rule (SUB), the rules for eliminating pair types ($\odot E$), sum types ($\odot +$), iso-recursive types ($\mu\eta E$), and function types ($\rightarrow E$), and the rule for typing let expressions (LET) are standard [3]. The add\_noise, $M$ expression is given type $!_{\infty}\mathbb{R}$ (ADD-NOISE), since the amount of noise added to $M$ suffices to hide any dependency on secret data. We will see later on that the noise parameter $s$ has to correspond to the replication index of the database type in order to achieve $\varepsilon$-DPD. The equality check between two values $M, N$ is well-typed with type $!_{\infty}\text{Bool}$ if the two values are of the same exponential type $!_{\infty}\tau$. The constraint on the replication factor is crucial to enforce distance preservation, as the possible results false and true of an equality check have distance $\infty$. Finally, the expression ref\_ creates a fresh reference and returns the pair of functions to read from and write into this reference: the type of this expression is thus obtained by pairing the types of the reader and writer.

**Soundness results.** In the following section we show the
main soundness results of our system. For the proofs we refer to the long version of this paper [18].

The proof of the differential privacy theorem relies on an opponent typability lemma, saying that all opponents are well-typed. As usual in type systems for cryptographic protocols [19], we require the opponent to be a closed expression that is annotated only with an untrusted type (Un in the literature, \(\tau \in \text{OPP}\) in our case), which characterizes the values that can be sent to and received from the opponent.

Note that we are not constraining the opponent, since typing annotations do not affect the semantics of expressions.

Our main theorem uses the following generalized definition of distributed differential privacy that considers constant terms \(D, D'\) of arbitrary type and arbitrary distance as inspired by Reed and Pierce [3].

**Definition 5 (Generalized \(\epsilon, \tau\)-DDP (\(\epsilon, \tau\)-DDP)).** \(P\) is \(\epsilon, \tau\)-differentially private iff for all constant terms \(D, D'\) of type \(\tau\) and all opponents \(O\),

\[
Pr[O(PD) \rightarrow^\ast 1] \leq e^{-\delta_\epsilon(D,D')} \cdot Pr[O(PD') \rightarrow^\ast 1]
\]

Theorem 1 below states that all well-typed expressions are \(\epsilon, \tau\)-differentially private. Note that we actually prove a more general property, parameterized by the noise added in the protocol execution.

**Theorem 1 ((\(\epsilon/k, \tau\))-Differential Privacy).** For all \(k \in \mathbb{R}^{>0}\), all types \(\tau\), and all closed expressions \(P\) such that the following conditions hold:

- the parameter of all noise addition primitives is set to \(k/\epsilon\) (i.e., they are of the form \(\text{add\_noise}_{k/\epsilon}(M)\))
- \(\emptyset : P : \tau \rightarrow \rho\) for some \(\rho \in \text{OPP}\)

\(P\) is \(\epsilon/k, \tau\)-differentially private.

We also remark that our type system can be used to enforce a strong secrecy property [22] based on probabilistic observational equivalence for expressions that do not contain any noise addition primitives, as formalized below.

**Corollary 1 (Strong Secrecy).** For all opponents \(O\), all types \(\tau\), and all closed expressions \(P\) such that the following conditions hold:

- \(P\) does not contain any noise addition primitives
- \(\emptyset : P : \tau \rightarrow \rho\) for some \(\rho \in \text{OPP}\)

we have that \(Pr[O(PD) \rightarrow^\ast 1] = Pr[O(PD') \rightarrow^\ast 1]\)

Finally, one might wonder how this approach scales to multiple protocol sessions. For example, let us consider a protocol that is given type \(\tau \rightarrow \rho\), where \(\rho \in \text{OPP}\) in which the parameter of all noise addition primitives is \(1/\epsilon\). By Theorem 1, this protocol achieves \(\epsilon, \tau\)-DDP. We might want to allow this protocol to be executed \(i\) times. Intuitively, such a multisession protocol will have type \(!_i \tau \rightarrow \rho\), since the secret database will have to be accessed \(i\) times, and will thus be \(\epsilon, !_i \tau\)-differentially private. This means that the ratio between the probability of outputting 1 when the protocol is initialized with database \(D\) and the probability of outputting 1 when the protocol is initialized with \(D'\) is bounded by \(e^{e^{-\delta_\epsilon(D,D')}}\). By the distance definition, this is equivalent to \(e^{\epsilon^{-\delta_\epsilon(D,D')}}\). This means that the multisession protocol is \((\epsilon, i), \tau\)-differentially private. Intuitively, privacy losses are summed up across multiple sessions (or queries), as stated by the principle of composition formulated by McSherry [25].

**VI. ALGORITHMIC TYPING**

The treatment of type bindings, as formulated in Section V, results in several non-deterministic rules. This non-determinism complicates the implementation of an efficient decision procedure. In the following section, we present a sound and complete algorithmic version of the type system.

**Algorithmic typing rules.** It is well-known that standard sources of non-determinism like subtyping and typing constructors of sum or iso-recursive types can be resolved using type annotations. We refer to the long version for the details and instead focus on the distinctive source of non-determinism of our system, the splitting of typing environ-
We first type-check the left pair component \(x\) as input in order to type-check expression \(A\) of the second premise and so on. The resources that were not consumed in the derivation of that premise are then returned and used in the derivation rule; the resources that were not consumed in the derivation search. Intuitively, the algorithmic variant of a typing rule type system is inspired by the work of Cervesato et al. [26] and allows us to type-check variables (\(V\)) such that

\[ \Gamma \vdash x : \tau \]

which is sound and complete. Intuitively, the algorithmic variant of the typing rules for variables (\(V\)) makes sense, which is much harder to deal with in an algorithmic way.

The core idea underlying the algorithmic version of the type system is inspired by the work of Cervesato et al. [26] on efficient resource management for linear logic proof search. Intuitively, the algorithmic variant of a typing rule that relies on the splitting of the typing environment among its premises proceeds as follows: all resources (i.e., type bindings) are first used to prove the first premise of a typing rule; the resources that were not consumed in the derivation of that premise are then returned and used in the derivation of the second premise and so on.

Algorithmic typing judgements are of the form \(\Gamma \vdash_{\text{alg}} A : \tau; \Gamma'\), where \(\Gamma\) denotes the typing environment that is given as input in order to type-check expression \(A\) with type \(\tau\) and \(\Gamma'\) denotes the environment entries that were not consumed in this derivation and can be used to prove further subgoals.

We note that every typing judgment of the form \(\Gamma \vdash A : \tau\) is matched by an algorithmic counterpart of the form \(\Gamma \vdash_{\text{alg}} A : \tau; \Gamma'\), which is sound and complete. Intuitively, this means that if \(\Gamma \vdash_{\text{alg}} A : \tau; \Gamma'\) then \(\Gamma \vdash A : \tau\) (soundness). Furthermore, if \(\Gamma \vdash A : \tau\) then there exists \(\Gamma'\) such that \(\Gamma \vdash_{\text{alg}} A : \tau; \Gamma'\) (completeness). We demonstrate the approach on the following algorithmic variants of the typing rules for variables (\(\text{VAR ALG}\)) and pairs (\(\otimes I \text{ ALG}\)):

\[
\begin{align*}
\text{VAR ALG} \\
\Gamma \vdash \phi & \quad x ::_{\phi} \phi \in \Gamma \quad k \geq 1 \\
\Gamma' &= \Gamma \{ x ::_{k-1} \phi / x ::_{k} \phi \} \\
\Gamma \vdash_{\text{alg}} x ::_{1} \phi &; \Gamma'
\end{align*}
\]

\[
\otimes I \text{ ALG} \\
\Gamma \vdash_{\text{alg}} M_1 : \tau_1; \Gamma' \quad \Gamma' \vdash_{\text{alg}} M_2 : \tau_2; \Gamma'' \\
\Gamma \vdash_{\text{alg}} (M_1, M_2) ::_{1} (\tau_1 \otimes \tau_2); \Gamma''
\]

A variable \(x\) can be type-checked with type \(1 ::_1 \phi\) under typing environment \(\Gamma\) if \(x\) is bound to core type \(\phi\) with replication index \(k \geq 1\) in \(\Gamma\). In the returned environment \(\Gamma'\) that can be used to type-check further subgoals, \(x\) is instead bound to \(\phi\) with reduced replication index \(k - 1\).

The rule \(\otimes I \text{ ALG}\) for pairs \((M_1, M_2)\) exemplifies the treatment of environment splitting: the complete typing environment \(\Gamma\) is used to type-check \(M_1\) with type \(\tau_1\). The resulting (possibly reduced) environment \(\Gamma'\) is then used to type-check \(M_2\) with type \(\tau_2\), which upon success returns the final remaining environment \(\Gamma''\) and allows us to type-check the pair \((M_1, M_2)\) with type \(1 :: (\tau_1 \otimes \tau_2)\).

**Example 4.** As an example, take \(\Gamma = x ::_{1} \text{ lg R}, A = (x, x)\), and \(\tau = 1 ::_{1} \text{ lg R}\). The algorithmic typing derivation to type-check \(A\) with type \(\tau\) under \(\Gamma\) looks as follows:

\[
\begin{align*}
\Gamma \vdash_{\text{alg}} x ::_{1} \text{ lg R} &; x ::_{1} \text{ lg R} \\
\Gamma \vdash_{\text{alg}} x ::_{1} \text{ lg R} &; x ::_{1} \text{ lg R} \\
\Gamma \vdash_{\text{alg}} (x, x) ::_{1} (\text{ lg R} \otimes \text{ lg R}) &; x ::_{1} \text{ lg R}
\end{align*}
\]

We first type-check the left pair component \(x\) with type \(1 ::_{1} \text{ lg R}\) under environment \(x ::_{1} \text{ lg R}\) and return the remaining environment \(1 ::_{1} \text{ lg R}\), which is then used to type-check the right pair component \(x\) again with type \(1 ::_{1} \text{ lg R}\). The pair is thus assigned type \(1 :: (\text{ lg R} \otimes \text{ lg R})\) and the remaining resources \(x ::_{1} \text{ lg R}\) are returned.

**Main results.** Below we state the completeness and soundness result of the algorithmic variant of our type system.

We write \(\langle A \rangle\) to denote the expression obtained by removing the typing annotations from \(A\).

**Theorem 2** (Completeness and Soundness). For every \(\Gamma, A,\) and \(\tau\), the following conditions hold:

1) If \(\Gamma \vdash A : \tau\) then there exist \(\Gamma', A'\) such that \(\Gamma \vdash_{\text{alg}} A' : \tau; \Gamma'\) and \(A = \langle A' \rangle\).

2) If \(\Gamma \vdash_{\text{alg}} A : \tau; \Gamma'\) then there exists \(\Gamma''\) such that \(\Gamma'' \vdash \langle A \rangle : \tau\) and \(\Gamma = \Gamma' + \Gamma''\).

**VII. A SEALING-BASED CRYPTO LIBRARY**

Originally, Morris [27] proposed the notion of dynamic sealing as a mechanism to protect program modules. While defining the semantics for a \(\lambda\)-calculus with dynamic sealing, Sumii and Pierce [28] later on observed a close correspondence with symmetric encryption. Bengtson et al. [19] showed how to encode a sealing-based cryptographic library for RCF using pairs, functions, references, and lists. In the following we propose a sealing-based cryptographic library for randomized symmetric cryptography that is well-typed in our type system.

**Standard sealing-based libraries.** The core idea of using seals to model cryptography is the following: a global reference is used to store a list of message-ciphertext pairs. This reference can only be accessed via the sealing and unsealing functions. The plaintext that is to be encrypted is paired with a fresh value, which represents the ciphertext. This message-ciphertext pair is added to the list (sealing). To decrypt a ciphertext, the latter is looked up in the list. If the ciphertext is in the list, then the corresponding message is returned (unsealing).

The symmetric key consists of the pair of the sealing and unsealing function. As shown by Bengtson et al. [19], we can model public-key cryptography (and other primitives) in a similar way: the sealing function corresponds to the encryption key, the unsealing function corresponds to the decryption key. For more details we refer to the long version [18].

**Linear sealing-based library.** As we have shown in Section IV, the crucial ingredient to enforce differential privacy is the linear usage of resources. In a distributed setting, one has to make sure that the sensitive (linear) data that is exchanged over the network does not get duplicated and processed more than once. Otherwise a single entry in the (distributed) database could have a huge, and not statically predictable, impact on the final result. Since we cannot prevent the opponent from duplicating public ciphertexts,
we must enforce that the content of each ciphertext cannot be processed more than once.

This can be achieved by encoding the sealing mechanism using a linear reference to store the list of message-ciphertext pairs. The resulting sealing function behaves as in the non-linear setting. In the unsealing function, the list is automatically removed from the referenced memory after being read. The list is then searched for the message-ciphertext pair. If the pair is contained in the list, the function will return the message and store the list pruned of the pair back into the reference. Thus, further decryptions of the same ciphertext are no longer possible. If the pair is not found, the complete list is stored back into the reference.

Soundness of the linear sealing-based library. Backes et al. [29] proved the computational soundness of a standard sealing-based library, establishing a semantic link between sealing-based cryptographic libraries and traditional Dolev-Yao constructor-based libraries. Here we show that our linear sealing-based cryptographic library can be proven sound with respect to a standard sealing-based library.

Intuitively, we have to make sure that the content of each ciphertext is not processed more than once, that is, the protocol is secure against replay attacks. There exist many standard techniques to defend against replay attacks, e.g., nonce-handshakes or session keys. We generalize this concept to the notion of guarded decryption. Intuitively, we let the cryptographic library contain multiple guarded decryption functions \texttt{dec&check}_i, which replace the original decryption functions. A guarded decryption function takes as input the decryption key and the ciphertext as well as an additional guard. It then unseals the ciphertext, applies a boolean check to the content of the ciphertext and the guard and only returns the plaintext if the check succeeded (otherwise, the plaintext is stored back into the seal). For instance, nonce-handshakes can be encoded in terms of a corresponding guarded decryption function \texttt{dec&check}$_{\text{nonce}}$ that takes the nonce as a guard and performs an equality check between the guard and the nonce contained in the ciphertext. We will introduce another guarded decryption function in the case study (cf. Section VIII).

In order to show the soundness of the linear sealing-based library, we have to put a restriction on the guarded decryption functions and on the usage of the guards in the protocol code. Intuitively, we say that a cryptographic library is valid if for each ciphertext \( c \) and key \( k \), there exists at most one \( i \) and one guard \( g \) such that \( \text{dec&check} \_i \_k \_c \_g \) succeeds (guard uniqueness). We also say that a program \( P \) is \( \mathcal{L}^{\text{crypto}} \)-valid if for every opponent \( O \) and every \( i \), \( O(\mathcal{L}^{\text{crypto}}; P) \) never calls \( \text{dec&check} \_i \_c \) more than once with the same guard (guard usage linearity). Guard uniqueness is a semantic property of the guarded decryption functions, which is trivial in all typical usages (e.g., for each ciphertext, there exists only one nonce for which the nonce check succeeds) and can thus be easily checked by hand. Guard usage linearity can be checked separately using existing static analysis techniques, e.g., \( \text{F}^* \) - a linear type system for implementations developed by Swamy et al. (into which RCF can be encoded) [30] - or LF7, a recently proposed type system for enforcing linear authorization policies in RCF [31]. Although we could have embedded one of these techniques in our type system, we intentionally factored out the problem of preventing replay attacks to keep the presentation clean and simple and to focus on the novel concepts introduced in this paper. For the formalization of guard uniqueness and guard usage linearity as well as information on how to enforce guard usage linearity using LF7, we refer to the long version [18].

In the following, we write \( \mathcal{L}^{\text{crypto}} \) to denote the standard sealing-based library, which is essentially obtained from \( \mathcal{L}^{\text{lin}} \) by replacing destructive references with non-destructive references. We can finally state the soundness result for our linear-based cryptographic library.

**Theorem 3** (Symbolic soundness of the cryptographic library). Let \( \mathcal{L}^{\text{crypto}} \) be a valid cryptographic library and \( P \) be a \( \mathcal{L}^{\text{crypto}} \)-valid program. For all opponents \( O \) and values \( M \), \( O(\mathcal{L}^{\text{crypto}}; P) \rightarrow^* \_p \_M \) if and only if \( O(\mathcal{L}^{\text{crypto}}; P) \rightarrow^* \_p \_M \).

**VIII. Case Study**

In this section we demonstrate the usefulness of our approach by analyzing a recently proposed protocol for non-tracking web analytics (NTWA) by Akkus et al. [17]. The system promises both differential privacy guarantees to the clients visiting a webpage and good quality analytics to the publishers of a webpage. It requires no additional authority that is not present in current web analytics scenarios.

Although the authors manually prove that the deployed noise mechanism that protects aggregated user data is indeed differentially private and they show how the system prevents different attacks, there is no formal proof that the overall system provides distributed differential privacy guarantees to the clients. In fact, it is not clear in general how to transfer the privacy results of the noise mechanism to establish privacy results for the overall protocol.

In the following, we give a brief description of the NTWA system and show the results of our analysis that include a newly discovered attack, a fix for the protocol, and the verified differential privacy result for the revised system.

**System overview.** The NTWA system comprises the three entities that can be found in standard web analytics scenarios: clients, publishers, and data aggregators. Intuitively, the provider of a website (publisher) uses a third party web analytics service (data aggregator) to gain aggregated information about the users visiting the site (clients). In today’s systems, the aggregator collects user information by tracking
the client’s behavior across the web and constructing a user profile, which poses a threat to the client’s privacy. The NTWA system follows a different approach by assuming the clients voluntarily answer queries about their personal data in exchange for better privacy guarantees.

The query mechanism of the NTWA system is depicted in Table I (cf. Section II). Intuitively, the client stores its personal information in a local database (e.g., demographics and browsing history). When visiting a webpage (step 1 in Table I), a client will be asked to answer some queries (step 2). These queries are stored at a fixed URL on the webpage. They may come from either the publisher or the aggregator and can be rather complex (e.g., SQL) but only allow for ‘yes’ or ‘no’ answers, e.g., “Are you between 35 and 50 years old?” The client encrypts its answers to these queries with the public key of the data aggregator (step 3) before sending them to the publisher, that acts as a proxy (step 4). The publisher collects answers from multiple clients and creates some fake answers (noise), encrypts them, and mixes the real and fake answers (step 5), which it then forwards to the aggregator (step 6). The data aggregator decrypts all the received query results, adds them together, adds some own noise to the results (step 7), and sends the signed “double noisy” results back to the publisher (step 8). The publisher subtracts its own noise from the received analytics to obtain its final aggregated result (step 9).

Overall, both the publisher and the data aggregator obtain analytics about the clients that visited the publisher’s webpage, but both results are not exact: the publisher cannot remove the noise that the aggregator added and vice-versa. As the authors show, the noise addition mechanisms they employ (variations of the Laplace Noise addition) are both differentially private, thus, intuitively, one should not be able to draw conclusions about a client’s personal database given the aggregated and noisy query results.

The trust assumptions underlying the NTWA protocol are as follows: (i) The client is trusted; (ii) the publishers can be selfishly malicious; (iii) the aggregator is honest-but-curious (HbC); (iv) the aggregator and the publisher do not collude, which is crucial to prevent the removal of the “double noise”.

We finally mention that the complete NTWA system also includes an auditing mechanism that is used to detect malicious publisher behavior, like dropping client answers. This mechanism is not necessary to achieve differential privacy and thus we omit it from our analysis, although it is useful in practice to detect attacks and identify malicious parties. For a more comprehensive overview of the system we refer to the original paper [17].

**Discovering the attack.** As we have shown in Section VII, the soundness of our approach relies on the linear usage of each ciphertext content. However, the NTWA protocol does not impose any restriction on the decryption of the encrypted client answers performed by the aggregator, thus allowing the same ciphertext to be decrypted multiple times, which leads to the replay attack described in Section II.

**The fix.** We propose the following minor modification to the protocol to prevent the snapshot attack. As we have seen, the problem is that one answer of a client may influence the final tally multiple times. We note that each query is associated with a query ID and a query end time after which answers to that query will no longer be processed. To ensure that each encrypted client answer is processed only once, we require the aggregator to perform the following steps: For each query identifier the aggregator stores the encrypted answers it received in response to that query in a duplicate-free list. At the query end time, all ciphertexts associated to a query will be decrypted and the aggregator will check that the query identifier inside the client’s answer corresponds to the expected one. If so, it will add the client’s answer to the tally and proceed as expected, if not, it will discard the answer. In other words, the decryption is guarded and the guard consists of the query identifier paired with the ciphertext. Note that this requires a public encryption scheme that does not allow for re-encryption. After the query end time, the stored ciphertexts will be discarded.

**Analysis of the revised protocol.** The analysis of the protocol has to take two different threat-levels into account: the protocol has to be secure both against a network-level attacker and against compromised parties, such as a malicious publisher or an HbC data aggregator. As usual in protocol analysis, the fact that an opponent has control over the network can be modeled by writing the protocol participants as cascaded functions that can be called and scheduled by the opponent at will. These functions take messages sent over the network as arguments when called by the opponent and return the messages they would otherwise output on the network. The potential compromise of the non-collaborating publisher and aggregator can be modeled by implementing two versions of the protocol: The former models the attack scenario in which the publisher is assumed to be honest and the aggregator to be HbC, the latter assumes an honest aggregator and a selfishly malicious publisher. In the former, the aggregator follows the protocol but leaks all data to the attacker, while in the latter the publisher role is directly played by the attacker.

While it is easy to verify the protocol under the assumption that all participants are honest and differential privacy has to be proven only with respect to a network-level opponent, verifying the protocol for an honest publisher and HbC data aggregator is currently out of the scope of our analysis technique. This is due to the fact that the noise mechanism employed by the publisher (Laplace noise with resampling) is a mechanism that gives only a weaker form of $e, \delta$-approximate differential privacy. Extending our type system to enforce approximate differential privacy is an interesting direction for future work.
is considered malicious and the data aggregator is honest. When analyzing the original NTWA protocol under this trust assumption we discovered the snapshot attack. We modeled the fixed protocol using a guarded decryption function $\text{dec}_{\text{NTWA}}$ for the encrypted client answers. The decryption function takes the pair of the ciphertext and the associated query identifier as guard and checks that the query identifier corresponds to the one given in the client’s answer. We successfully type-checked the fixed protocol, showing $\epsilon$-DDP against a malicious publisher and a network level opponent. For more details we refer to the long version [18].

IX. RELATED WORK

The formal verification of differential privacy has recently received increasing attention by the academic community. Barthe et al. have presented CertiPriv [5], a machine-checked framework for reasoning about differential privacy built on top of the Coq proof assistant. This framework nicely complements our approach, allowing one to derive formal guarantees of differential privacy for a variety of sanitization mechanisms, such as the one based on Laplace noise, whose correctness is instead assumed in our approach. CertiPriv, however, has not been used to reason about complex cryptographic protocols and network-level attacks, and proofs are not automated. Tschantz et al. [32] showed how to verify differential privacy properties based on I/O-automata. They focus on the usage of differentially private sanitization mechanisms within interactive systems, but they do not explicitly consider cryptographic protocols. Recently, Chaudhuri et al. [33] have introduced a generic robustness property for programs that encompasses sensitivity and can be statically enforced even for programs featuring complex branching structures and loops. This could be useful to further enhance the expressivity of our framework.

While the present paper focuses on the notion of differential privacy, we would like to mention some recent works that discuss some limitations of differential privacy [34], [35] and propose alternative notions of privacy for queries on statistical databases, such as relaxations of differential privacy [36], noiseless definitions [37], and zero-knowledge based definitions for social networks [38].

Finally, since the seminal work by Abadi on “secrecy by typing in security protocols” [39], type systems acquired growing popularity in the analysis of cryptographic protocols and their implementations, and they have been applied to statically enforce secrecy definitions based on reachability properties [20], [40], strong secrecy [22] properties based on observational equivalence relations, as well as privacy [41], [42], authenticity [43]–[48] and authorization policies [?], [19], [24], [31], [49], [50]. None of these type systems, however, enforces quantitative secrecy properties.

X. CONCLUSION AND FUTURE WORK

This work introduces the first mechanized verification technique for distributed differential privacy. Our framework comprises a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders, and a linear, distance-aware type system to verify this property in cryptographic protocol implementations. A sound and complete algorithmic variant of the type system allows for mechanizing our analysis technique. To the best of our knowledge, this is first automated verification technique for cryptographic protocols that supports a quantitative secrecy property. We have evaluated our system on a protocol for non-tracking web analytics and discovered a new attack. We proposed and verified a revised version of the protocol.

We are currently implementing a type-checker, based on the algorithmic variant of our type system. We are investigating how to leverage recent results on type inference for linearly typed functional languages [51] in order to develop a type inference algorithm to reduce typing annotations.

We are also extending our cryptographic library to other schemes, such as homomorphic encryptions, and the type system to other sanitization mechanisms, such as binomial-based distributed noise generation [6], which would allow us to enlarge the scope of our verification framework to further protocol classes (e.g., [13]).

Finally, we are currently investigating how to extend our framework to deal with sanitization mechanisms that provide $\epsilon, \delta$-approximate differential privacy.

Acknowledgments. This work was supported by the German research foundation (DFG) through the Emmy Noether program and the Cluster of Excellence on Multimodal Computing and Interaction (MMCI), and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA). We thank the reviewers for their helpful comments.

References

The exponential mechanism. There exist many scenarios in which the query result is non-numerical (e.g., queries returning strings or trees) and adding noise leads to nonsensical results or is not well-defined. McSherry and Talwar [16] proposed a general technique to optimize the quality (and exactness) of a query result while still preserving $\epsilon$-differential privacy. Their so-called exponential mechanism works on queries on databases $D$ of some type $\mathcal{D}$ that are expected to return a query result $a$ of an arbitrary type $\mathcal{R}$ for which a base measure $\beta$ exists. Furthermore, it assumes the existence of a utility function $q : (\mathcal{D} \times \mathcal{R}) \rightarrow \mathbb{R}$, which assigns a real valued score to each possible input-output pair $(D, a)$, thereby measuring the quality of the result $a$ with respect to input $D$. The higher the score, the better (e.g., more exact) the result. The goal of the mechanism $\varepsilon^*_{q}(D)$ is to output the “best” possible result $a \in \mathcal{R}$, while enforcing differential privacy.

**Definition 6** (Exponential Mechanism [16]). For all $q : (\mathcal{D} \times \mathcal{R}) \rightarrow \mathbb{R}$ and all base measures $\beta$ over $\mathcal{R}$ the randomized exponential mechanism $\varepsilon^*_{q}(D)$ for $D \in \mathcal{D}$ is defined as

$$
\varepsilon^*_{q}(D) := \text{return } a \in \mathcal{R} \text{ with probability proportional to } e^{q(D,a)} \cdot \beta(a).
$$

McSherry and Talwar [16] show that this definition captures the entire class of differential privacy mechanisms and give an encoding of Laplace noise addition by choosing an appropriate utility function $q$. The link between the exponential mechanism and differential privacy is shown in the privacy theorem below. Here $\Delta q$ is defined as the largest possible difference in the utility function when applied to two inputs that differ only on a single user’s value [16].

**Theorem 4** (Privacy [16]). $\varepsilon^*_{q}(D)$ gives $2\epsilon \Delta q$-differential privacy.

**Extending the syntax and semantics.** We now show how to extend the syntax and semantics of our calculus to include the exponential mechanism $\varepsilon^*_{q}(D)$ for utility functions $q$ with known $\Delta q$. The following primitive is added to the set of expressions:

$$
\text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R} \text{ } M.
$$

It is annotated with the parameter $s$ and the appropriate utility function $q : (b_1 \times b_2) \rightarrow \mathbb{R}$, where both the domain $b_1$ and the range $b_2$ of the query are base types in our type system. Its non-deterministic semantics is defined similarly to that of the add\textunderscore noise primitive:

$$
[S, \text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R} \text{ } c_1] \quad \text{Exp} \quad (q; (b_1 \times b_2)) \rightarrow \mathbb{R} \text{, } c_1, c_2, s) \quad \rho \quad [S, c_2],
$$

where $p = Pr[\varepsilon^*_s(c_1) = c_2]$ and $c_1 : b_1 \in \Sigma$. Intuitively, $\text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R} \text{ } c_1$ reduces to some constant value $c_2$ in the query range, where $c_2$ is drawn according to the distribution $\varepsilon^*_s(c_1)$. Note that the density of $\varepsilon^*_s(D)$ is defined as

$$
Pr[x] = \frac{e^{s \cdot q(D,x)} \cdot \beta(x)}{\int e^{s \cdot q(D,x)} \cdot \beta(x)dx}.
$$

Here, the safety parameter $s$ has the same function as its counterpart in $\text{Lap}_s$. Intuitively, one would assume it to be set to $s := \epsilon$ but since we know that $\varepsilon^*_s(D)$ gives $2\epsilon \Delta q$-differential privacy (by Theorem 4) and we are overall interested in a guarantee of $\epsilon / k$, $\tau$-differential privacy we will choose $s$ to be set to $s := \epsilon / (k / \Delta q)$.

**Extending the type system.** To type-check the exponential mechanism primitive, the following rule is added to the type system

$$
\text{EXP-MECH}
$$

$$
\Gamma \vdash M : !_1 b_1
$$

$$
\Gamma \vdash \text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R} \text{ } M : !_\infty b_2.
$$

Here, the argument $M$ of the exponential mechanism that is parameterized by the utility function $q : (b_1 \times b_2) \rightarrow \mathbb{R}$ must be of the domain type $b_1$ of the query with replication factor $!_1$ (since it might be private). The sanitized result is then given the query range type $b_2$, replicated by a factor of $!_\infty$, since it may be published.

**Adapting the theorem.** We need to modify Theorem 1 to accommodate the occurrences of the exponential mechanism primitives. The revised theorem is given below and requires the $\text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R}$ primitives to be annotated with an appropriate security parameter $s$ to guarantee $\epsilon / k$, $\tau$-differential privacy.

**Theorem 1** ($($/$k$, $\tau$)-Differential Privacy). For all $k \in \mathbb{R}^{>0}$, all types $\tau$, and all closed expressions $P$ such that the following conditions hold:

- the parameter of all noise addition primitives is set to $k/\epsilon$ (i.e., they are of the form add\textunderscore noise$^s_{k/\epsilon} \text{ } M$)

- the parameter of all exponential mechanism primitives for a utility $q$ is set to $s := \epsilon / (k / \Delta q)$ (i.e., they are of the form $\text{exp\textunderscore mech}^s_{q; (b_1 \times b_2)} \rightarrow \mathbb{R} \text{ } M$)

- $\emptyset \vdash P : \tau \rightarrow \rho$ for some $\rho \in \text{OPPP}$

$P$ is $\epsilon / k$, $\tau$-differentially private.

The proofs of the soundness of the extended type system do not depend on any specific property of the exponential mechanism, just on the respective privacy theorem (Theorem 4). Indeed, we believe that our framework can easily be extended to many other privacy mechanisms.