# PROCHLO: Strong Privacy for Analytics in the Crowd

Andrea Bittau* Úlfar Erlingsson* Petros Maniatis* Ilya Mironov* Ananth Raghunathan*
David Lie‡ Mitch Rudominer° Ushasree Kode° Julien Tinnes° Bernhard Seefeld°

*Google Brain ‡Google Brain and U. Toronto °Google



## Abstract

The large-scale monitoring of computer users' software activities has become commonplace, e.g., for application telemetry, error reporting, or demographic profiling. This paper describes a principled systems architecture—*Encode, Shuffle, Analyze* (ESA)—for performing such monitoring with high utility while also protecting user privacy. The ESA design, and its PROCHLO implementation, are informed by our practical experiences with an existing, large deployment of privacy-preserving software monitoring.

With ESA, the privacy of monitored users' data is guaranteed by its processing in a three-step pipeline. First, the data is *encoded* to control scope, granularity, and randomness. Second, the encoded data is collected in batches subject to a randomized threshold, and blindly *shuffled*, to break linkability and to ensure that individual data items get "lost in the crowd" of the batch. Third, the anonymous, shuffled data is *analyzed* by a specific analysis engine that further prevents statistical inference attacks on analysis results.

ESA extends existing best-practice methods for sensitive-data analytics, by using cryptography and statistical techniques to make explicit how data is elided and reduced in precision, how only common-enough, anonymous data is analyzed, and how this is done for only specific, permitted purposes. As a result, ESA remains compatible with the established workflows of traditional database analysis.

Strong privacy guarantees, including differential privacy, can be established at each processing step to defend against malice or compromise at one or more of those steps. PROCHLO develops new techniques to harden those steps, including the *Stash Shuffle*, a novel scalable and efficient oblivious-shuffling algorithm based on Intel's SGX, and new applications of cryptographic secret sharing and blinding. We describe ESA and PROCHLO, as well as experiments that validate their ability to balance utility and privacy.



*SOSP '17,* October 28, 2017, Shanghai, China



ACM ISBN 978-1-4503-5085-3/17/10…

DOI: https://doi.org/10.1145/3132747.3132769
Reprinted from SOSP '17,, [Unknown Proceedings], October 28, 2017, Shanghai, China, pp. 1–19.

## 1. Introduction

Online monitoring of client software behavior has long been used for disparate purposes, such as measuring feature adoption or performance characteristics, as well as large-scale error-reporting [34]. For modern software, such monitoring may entail systematic collection of information about client devices, their users, and the software they run [17, 60, 69]. This data collection is in many ways fundamental to modern software operations and economics, and provides many clear benefits, e.g., it enables the deployment of security updates that eliminate software vulnerabilities [62].

For such data, the processes, mechanisms, and other means of privacy protection are an increasingly high-profile concern. This is especially true when data is collected automatically and when it is utilized for building user profiles or demographics [21, 69, 71]. Regrettably, in practice, those concerns often remain unaddressed, sometimes despite the existence of strong incentives that would suggest otherwise. One reason for this is that techniques that can guarantee privacy exist mostly as theory, as limited-scope deployments, or as innovative-but-nascent mechanisms [5, 7, 25, 28].

We introduce the *Encode, Shuffle, Analyze* (ESA) architecture for privacy-preserving software monitoring, and its PROCHLO implementation.[1] The ESA architecture is informed by our experience building, operating, and maintaining the RAPPOR privacy-preserving monitoring system for the Chrome Web browser [28]. Over the last 3 years, RAPPOR has processed up to billions of daily, randomized reports in a manner that guarantees local differential privacy, without assumptions about users' trust; similar techniques have since gained increased attention [6, 7, 70, 74]. However, these techniques have limited utility, both in theory and in our experience, and their statistical nature makes them ill-suited to standard software engineering practice.

Our ESA architecture overcomes the limitations of systems like RAPPOR, by extending and strengthening current best practices in private-data processing. In particular, ESA enables any high-utility analysis algorithm to be compatible with strong privacy guarantees, by appropriately building on users' trust assumptions, privacy-preserving randomization,

[1] PROCHLO combines privacy with the Greek word ὄχλος for crowds.

and cryptographic mechanisms—in a principled manner, encompassing realistic attack models.

With ESA, software users' data is collected into a database of encrypted records to be processed only by a specific *analysis*, determined by the corresponding data decryption key. Being centralized, this database is compatible with existing, standard software engineering processes, and its analysis may use the many known techniques for balancing utility and privacy, including differentially-private release [27].

The contents of a materialized ESA database are guaranteed to have been *shuffled* by a party that is trusted to remove implicit and indirect identifiers, such as arrival time, order, or originating IP address. Shuffling operates on batches of data which are collected over an *epoch* of time and reach a minimum threshold cardinality. Thereby, shuffling ensures that each record becomes part of a *crowd*—not just by hiding timing and metadata, but also by ensuring that more than a (randomized) threshold number of reported items exists for every different data partition forwarded for analysis.

Finally, each ESA database record is constructed by specific reporting software at a user's client. This software ensures that records are *encoded* for privacy, e.g., by removing direct identifiers or extra data, by fragmenting to make the data less unique, or by adding noise for local differential privacy or plausible deniability. Encoded data is transmitted using nested encryption to ensure that the data will be shuffled and analyzed only by the shuffler and analyzer in possession of the corresponding private keys.

ESA protects privacy by building on users' existing trust relationships as well as technical mechanisms. Users indicate their trust assumptions by installing and executing client-side software with embedded cryptographic keys for a specific shuffler and analyzer. Based on those keys, the software's reporting features encode and transmit data for exclusive processing by the subsequent steps. Each of these processing steps may independently take measures to protect privacy. In particular, by adding random noise to data, thresholds, or analysis results, each step may guarantee differential privacy (as discussed in detail in §3.5).

Our contributions are as follows:

- We describe the ESA architecture, which enables high-utility, large-scale monitoring of users' data while protecting their privacy. Its novelty stems from explicit trust assumptions captured by nested encryption, guarantees of anonymity and uncertainty added by a shuffler intermediary, and a pipeline where differential privacy guarantees can be independently added at each stage.
- We describe PROCHLO, a hardened implementation of ESA that uses SGX [36], showing strong privacy guarantees to be compatible with efficiency, scalability, and a small trusted computing base. We design, implement, and evaluate a new oblivious-shuffling algorithm for SGX, the *Stash Shuffle*, that scales efficiently to our target data sizes, unlike previous algorithms. We also introduce new cryptographic primitives for secret-shared, blind data thresholding, reducing users' trust requirements.
- We evaluate PROCHLO on two representative monitoring use cases, as well as a collaborative filtering task and a deep-learning task, in each case achieving both high-utility and strong privacy guarantees in their analysis.

On this basis we conclude that the ESA architecture for monitoring users' data significantly advances the practical state-of-the-art, compared to previous, deployed systems.

## 2. Motivation and Alternatives

The systems community has long recognized the privacy concerns raised by software monitoring, and addressed them in various ways, e.g., by authorization and access controls. Notably, for large-scale error reporting and profiling, it has become established best practice for systems software to perform data reduction, scrubbing, and minimization—respectively, to eliminate superfluous data, remove unique identifiers, and to coarsen and minimize what is collected without eliminating statistical signals—and to require users' opt-in approval for each collected report. For example, large-scale Windows error reporting operates in this manner [34].

However, this established practice is ill-suited to automated monitoring at scale, since it is based solely on users' declared trust in the collecting party and explicit approval of each transmitted report, without any technical privacy guarantees. Perhaps as a result, there has been little practical deployment of the many promising mechanisms based on pervasive monitoring developed by the systems community, e.g., for understanding software behavior or bugs, software performance or resource use, or for other software improvements [13, 39, 46, 59, 61, 77]. This is ironic, because these mechanisms rely on common statistics and correlations—not individual users' particular data—and should therefore be compatible with protecting users' privacy.

To serve these purposes, the ESA architecture is a flexible platform that allows high-utility analysis of software-monitoring data, without increasing the privacy risk of users. For ease of use, ESA is compatible with, and extends, existing software engineering processes for sensitive-data analytics. These include data elimination, coarsening, scrubbing, and anonymization, both during collection and storage, as well as careful access controls during analysis, and public release of only privacy-preserving data [28].

### 2.1 A Systems Use Case for Software Monitoring

As a concrete motivating example from the domain of systems software, consider the task of determining which system APIs are used by which individual software application. This task highlights the clear benefits of large-scale software monitoring, as well as the difficulty of protecting users' privacy. Just one of its many potential benefits is the detection of seldom-used APIs and the cataloging of applications still

using old, legacy APIs, such that those APIs can be deprecated and removed in future system versions.

Naïvely collecting monitoring data for this task may greatly affect users' privacy, since both the identity of applications and their API usage profiles are closely correlated to user activities—including those illegal, embarrassing, or otherwise damaging, such as copyright infringement, gambling, etc. Indeed, both applications and the combinations of APIs they use may be unique, incriminating, or secret (e.g., for developers of their own applications). Therefore, to guarantee privacy, this task must be realized without associating users with the monitored data, without revealing secret applications or API usage patterns, and without creating a database of sensitive (or deanonymizable) data—since such databases are at risk for abuse, compromise, or theft.

This paper describes how this monitoring task can be achieved—with privacy—using the ESA architecture. However, we must first consider what is meant by privacy, and the alternative means of guaranteeing privacy.

### 2.2 Privacy Approaches, Experience, and Refinements

Generally, privacy is intuitively understood as individuals' socially-defined ability to control the release of their personal information [5,64]. As such, privacy is not easily characterized in terms of traditional computer security properties. For example, despite the integrity and secrecy of input data about individuals, social perceptions about those individuals may still be changed if the observable output of computations enables new statistical inferences [22].

***Privacy Approaches and Differential Privacy Definitions***

Myriad new definitions and techniques have been proposed for privacy protection—mostly without much traction or success. For example, the well-known property of *k-anonymity* prevents direct release of information about subgroups with fewer than $k$ individuals [65, 66]. However, since $k$-anonymity does not limit statistical inferences drawn indirectly, from subgroup intersections or differences, it cannot offer mathematically-strong privacy guarantees [32, 48].

In the last decade, the definitions of differential privacy have become the accepted standard for strong privacy [26, 27]. The guarantees of differential privacy stem from the uncertainty induced by adding carefully-selected random noise to individuals' data, to the intermediate values computed from that data, or to the final output of analysis computations. As a result, each-and-every individual may be sure that *all* statistical inferences are insensitive to their data; that is, their participation in an analysis makes no difference, to their privacy loss—even in the worst case, for the unluckiest individual. Specifically, the guarantees establish an $\varepsilon$ upper bound on the sensitivity in at least $1 - \delta$ cases; thus, an $(\varepsilon, \delta)$-private epidemiology study of smoking and cancer can guarantee (except, perhaps, with $\delta$ probability) that attackers' guesses about each participant's smoking habits or illnesses can change by a multiplicative $e^{\varepsilon}$ factor, at most.

A wide range of mechanisms have been developed for the *differentially-private release* of analysis results. Those mechanisms support analysis ranging from simple statistics through domain-specific network analysis, to general-purpose deep learning of neural networks by stochastic gradient descent [4, 27, 51]. Alas, most of these differentially-private mechanisms assume that analysis is performed on a trusted, centralized database platform that can securely protect the long-term secrecy and integrity of both individuals' data and the analysis itself. This is disconcerting, because it entails that any platform compromise can arbitrarily impact privacy—e.g., by making individuals' data public—and the risk of compromise seems real, since database breaches are all too common [41, 56].

Instead of adding uncertainty to analysis output to achieve differential privacy, an alternative is to add random noise to each individual's data, before it is collected. This can be done using techniques that provide local differential privacy (e.g., randomized response) in a manner that permits moderately good utility [7, 28, 70, 74]. By adopting this approach, the risk of database compromise can be fully addressed: the data collected from each individual has built-in privacy guarantees, which hold without further assumptions and require no trust in other parties.

***Experiences with Local Differential Privacy Mechanisms***

Local differential privacy is the approach taken by our RAPPOR software monitoring system for the Chrome Web browser [28]. Over the last 3 years, we have deployed, operated, and maintained RAPPOR versions utilized for hundreds of disparate purposes, by dozens of software engineers, to process billions of daily, randomized reports [16].

Regrettably, there are strict limits to the utility of locally-differentially-private analyses. Because each reporting individual performs independent coin flips, any analysis results are perturbed by noise induced by the properties of the binomial distribution. The magnitude of this random Gaussian noise can be very large: even in the theoretical best case, its standard deviation grows in proportion to the square root of the report count, and the noise is in practice higher by an order of magnitude [7, 28–30, 74]. Thus, if a billion individuals' reports are analyzed, then a common signal from even up to a million reports may be missed.

Because of their inherent noise, local differential privacy approaches are best suited for measuring the most frequent elements in data from peaky power-law distributions. This greatly limits their applicability—although they may sometimes be a good fit, such as for RAPPOR's initial purpose of tracking common, incorrect software configurations. For example, a thousand apps and a hundred APIs may be relevant to a task of measuring applications' API usage (§2.1). However, an infeasible amount of data is required to get a clear signal for each app/API combination with local differential privacy: an order-of-magnitude more than 100,000, squared, i.e., reports from one trillion individual users [30].

Somewhat surprisingly, if the set of reports can be partitioned in the right manner, the utility of RAPPOR (and similar systems) can be greatly enhanced by analyzing fewer reports at once. By placing correlated data into the same partitions, signal recovery can be facilitated, especially since the square-root-based noise floor will be lower in each partition than in the entire dataset.

In particular, the data required for the app/API example above can be reduced by two orders of magnitude, if the reported API is used to partition RAPPOR reports into 100 disjoint, separately-analyzed sets; for each separate API, only 100 million reports are required to find the top 1000 apps, by the above arithmetic (see also §5.2's experiment).

Unfortunately, such partitioning may greatly weaken privacy guarantees: differential privacy is fundamentally incompatible with the certain knowledge that an API was used, let alone that a particular individual used that API [22, 32]. Therefore, any such partitioning must be done with great care and in a way that adds uncertainty about each partition.

Another major obstacle to the practical use of locally-differentially-private methods—based on our experiences with RAPPOR—is the opaque, fixed, and statistical nature of the data collected. Not only does this prevent exploratory data analysis and any form of manual vetting, but it also renders the reported data incompatible with the existing tools and processes of standard engineering practice. Even when some users (e.g., of Beta or Developer software versions) have opted into reporting more complete data, this data is not easily correlated with other reports because it is not collected via the same pipelines. In our experience, this is a frustrating obstacle to developers, who have been unable to get useful signals from RAPPOR analysis in a substantial fraction of cases, due to the noise added for privacy, and the difficulties of setting up monitoring and interpreting results.

***Insights, Refinements, and Cryptographic Alternatives***

The fundamental insight behind the ESA architecture is that both of the above problems can be eliminated by collecting individuals' data through an intermediary, in a unified pipeline. This intermediary (the ESA shuffler) explicitly manages data partitions (the ESA crowds), and guarantees that each partition is sufficiently large and of uncertain-enough size. This is done by batching and randomized thresholding, which establishes differential privacy and avoids the pitfalls of $k$-anonymity [22, 32, 33, 48]. Furthermore, this intermediary can hide the origin of reports and protect their anonymity—even when some are more detailed reports from opt-in users—and still permit their unified analysis (e.g., as in Blender [6]). This anonymity is especially beneficial when each individual sends more than one report, as it can prevent their combination during analysis (cf. [70]).

ESA can be seen as a refinement of existing, natural trust relationships and best practices for sensitive data analytics, which assigns the responsibility for anonymity and randomized thresholding to an independently-trusted, standalone intermediary. ESA relies on cryptography to denote trust, as well as to strengthen protection against different attack models, and to provide privacy guarantees even for unique or highly-identifying report data. Those cryptographic mechanisms—detailed in the remainder of this paper—differ from the cryptography typically used to protect privacy for aggregated analysis (or for different purposes, like Vuvuzela or Riposte private messaging [19, 72]).

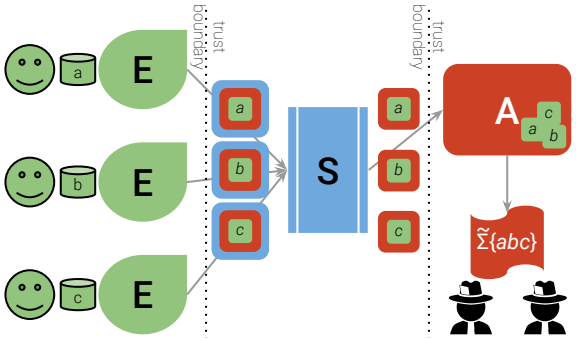


Figure 1: ESA architecture: Encode, shuffle, and analyze.

Other cryptography-based privacy-protection systems, such as PDDP, Prio, and Secure Aggregation [11, 15, 18], mostly share ESA's goals but differ greatly in their approach. By leveraging multiparty computations, they create virtual trusted-third-party platforms similar to the centralized PINQ or FLEX systems, which support differentially-private release of analysis results about user data [38, 52]. These approaches can improve user-data secrecy, but must rely on added assumptions, e.g., about clients' online availability, clients' participation in multi-round protocols, and attack models with an honest-but-curious central coordinator. Also, in terms of practical adoption, these systems require radical changes to engineering practice and share RAPPOR's obstacle of making user data overly opaque and giving access only to statistics.

In comparison, ESA is compatible with existing, unchanged software engineering practices, since the output from the ESA shuffler can be gathered into databases that have built-in guarantees of uncertainty and anonymity. Furthermore, ESA offers three points of control for finding the best balance of privacy and utility, and the best protections: local-differential privacy, at the client, randomized thresholding and anonymity, at the privacy intermediary, and differentially-private release, at the point of analysis.

## 3. The Encode-Shuffle-Analyze Architecture

The ESA architecture splits responsibility between *encoders*, *shufflers*, and *analyzers*, as shown in Figure 1.

Encoders run on the client devices of users, potentially as part of the software being monitored. They can transform the monitored data in a number of ways—in particular, by converting its encoding—and they also use nested encryption to guarantee which parties process it and in what order.

For privacy purposes, encoders control the scope, granularity, and randomness of the data released from clients; in particular, they can add enough random noise to provide users with strong guarantees, without trust assumptions [28].

Shufflers are a separate, standalone networked service that receives encrypted, encoded data. They are responsible for masking data origin and confounding data provenance by eliminating metadata (arrival time, order, originating IP address) and by shuffling (to thwart traffic analysis). Shufflers collect data into batches during a lengthy interval (e.g., one day), not only to eliminate implicit timing metadata, but also to ensure that every data item can get lost in a crowd of similar items. Large-enough batches are forwarded for analysis after undoing a layer of nested encryption. Shufflers should be trustworthy parties and (at least logically) completely independent of analysis services—although their collusion is considered in our attack model.

Analyzers are networked services that receive shuffled data batches, and are responsible for materializing and analyzing or releasing a database after undoing the innermost encryption. Their cryptographic key determines the specific analysis outcome and associated privacy protection (§3.1).

To illustrate, consider the app/API example from §2.1 (see §5 for detailed use cases). For this example, the ESA architecture enables high utility and privacy without requiring a deluge of user data. Since this analysis only seeks statistics about individual API uses in apps and not multi-API patterns, ESA encoding can transmit data as multiple, broken-apart bitvector fragments (§3.2), without necessarily adding random noise. This fragmentation breaks apart unique patterns that might identify users of rare applications and does not affect the utility of the intended statistical analysis.

ESA shuffling breaks any direct associations between data fragments of individual users, thereby hiding their data in the crowd (§3.3). It can also add uncertainty by randomly thresholding the data before forwarding to analysis. By using cryptographic secret-sharing (§4.2) and blinding (§4.3), ESA enables the collection of a database of app names for analysis, without revealing any unique apps. For hard-to-guess, hash-based app names, this guarantee can hold even for our strongest attack model below, where all parties, including the shuffler, collude against users (§5.2 is a concrete example).

Thanks to ESA, the combined data from app/API bitvector fragments need not be considered sensitive—even if no differential privacy was introduced by adding random noise during encoding. Therefore, this data is well-suited for analysis using standard tools, techniques, and processes. Even so, by adding minimal further noise at the ESA analyzer (§3.4), the analysis output may be subject to differentially-private release, at no real loss to utility.

### 3.1 Keys, Trust, and Attack Models

Users rely on the encoder's use of specific keys for nested encryption to specify who should receive monitoring data, what that data should contain, and how it should be processed. Thus, users state their trust assumptions implicitly using those keys. This trust may be well placed, for example, when the cryptographic keys are embedded in software that the user must trust (such as their operating system), and where those keys are associated with clear policies for the processing of the monitored data. The users' trust in those keys may also be misplaced, e.g., if the keys have been compromised, or if different parties collude. This provides a natural structuring of an attack model for the ESA architecture:

**Analyzer compromise** The analyzer is the primary attacker against which ESA defends. Its attacks can, in particular, link different data received, and correlate it with auxiliary information—for example, overlapping, public databases—in order to breach users' privacy.

**Shuffler compromise** The shuffler is assumed to be honest but curious. If compromised, it reveals the types of reports sent—when, in what frequency, from which IP addresses, in what data partition—but contents are still protected by the inner layer of nested encryption.

**Analyzer and shuffler collusion** If attackers control both the shuffler and the analyzer, they will see which users contribute what data—as if they had compromised a monitoring data collection service not using the ESA architecture. Even then, encoding may still provide some protections for users' privacy (e.g., as in RAPPOR [28]).

**Encoder compromise and collusion** In all cases, attackers may control most (but not all) encoders, and use them to send specially-crafted reports. In collusion with an analyzer or shuffler adversary, this enables more powerful attacks, e.g., where users' data is combined into a "crowd" comprising only Sybil data [24]. Also, this enables data-pollution attacks on analysis, for which creation of a materialized database or a hardened encoder may be partial remedies [1,47], as might be some multi-party computation schemes proposed before (e.g., Prio [18]). However, we do not consider data pollution or Sybil attacks further in this work.

**Public information** In all cases, attackers are assumed to have complete access to network communications, as well as analysis results, e.g., using them to perform timing and statistical inference attacks.

ESA can still provide differential privacy guarantees to users under this powerful attack model.

### 3.2 Encoder

Encoders run at users' clients, where they mediate upon the reporting of monitored data and control how it is transmitted for further processing. As described above, encoders are tied to and reflect users' trust assumptions.

Encoders locally transform and condition monitored data to protect users' privacy—e.g., by coarsening or scrubbing, by introducing random noise, or by breaking data items into

fragments—apply nested encryption, and transmit the resulting reports to shufflers, over secure channels. Encoders also help users' data become lost in a crowd, by marking transmitted reports with a *crowd ID*, upon which the ESA shuffler applies a minimum-cardinality constraint, preventing analysis of any data in crowds smaller than a threshold.

For most attack models, encoders can provide strong privacy guarantees even by fragmenting the users' data before transmission. For example, consider analysis involving users' movie ratings, like that evaluated in §5.5. Each particular user's set of movie ratings may be unique, which is a privacy concern. To protect privacy, encoders can fragment and separately transmit users' data; for example, the rating set $\{(m_0, r_0), (m_1, r_1), (m_2, r_2)\}$, may be encoded as its pairwise combinations $\langle (m_0, r_0), (m_1, r_1)\rangle$, $\langle (m_0, r_0), (m_2, r_2)\rangle$, and $\langle (m_1, r_1), (m_2, r_2)\rangle$, with each pair transmitted for independent shuffling. Similarly, most data that is analyzed for correlations can be broken up into fragments, greatly increasing users' privacy.

As an important special case, encoders can apply randomized response to users' data, or similarly add random noise in a way that guarantees local differential privacy. While useful, this strategy has drawbacks—as discussed earlier in §2.2. Other ESA encoders can offer significantly higher utility, while still providing differential privacy (albeit with different trust assumptions). For example, §4.2 discusses a PROCHLO encoder based on secret sharing.

### 3.3 Shuffler

The shuffler logically separates the analyzer from the clients' encoders, introducing a layer of privacy between the two. To help ensure that it is truly independent, honest-but-curious, and non-colluding, the shuffler may be run by a trusted, third-party organization, run by splitting its work between multiple parties, or run using trusted hardware. Such technical hardening is discussed in §4.1.

The shuffler performs four tasks: *anonymization*, *shuffling*, *thresholding*, and *batching*, described in turn.

As the first pipeline step after client encoders, shufflers have access to user-specific metadata about the users' reports: timestamps, source IP addresses, routing paths, etc. (Notably, this metadata may be useful for admission control against denial-of-service and data poisoning.) A primary purpose of shufflers is to strip all such implicit metadata, to provide anonymity for users' reports.

However, stripping metadata may not completely disassociate users from reports. In our attack model, a compromised analyzer may monitor network traffic and use timing or ordering information to correlate individual reports arriving at the shuffler with (stripped) data forwarded to the analyzer. Therefore, shufflers forward stripped data infrequently, in batches, and only after randomly reordering the data items.

Even stripped and shuffled data items may identify a client due to uniqueness. For example, a long-enough API bitvector may be uniquely revealing, and the application it concerns may be truly unique—e.g., if it was just written. An attacker with enough auxiliary information may be able to tie this data to a user and thereby break privacy (e.g., by learning secrets about a competitor's use of APIs). To prevent this, the shuffler can do thresholding and throw away data items from item classes with too few exemplars. That is the purpose of the crowd ID, introduced into the data item by the encoder: it allows the shuffler to count data items for each crowd ID, and filter out all items whose crowd IDs have counts lower than a fixed, per-pipeline threshold. In the API use case, the crowd ID could be some unique application identifier, ensuring that applications with fewer than, say, 10 contributed API call vectors are discarded from the analysis.

Shuffling and thresholding have limited value for small data sets. The shuffler batches data items for a while (e.g., a day) or until the batch is large enough, before processing.

### 3.4 Analyzer

The analyzer decrypts, stores, aggregates, and eventually retires data received from shufflers. Although sensitivity of data records forwarded by shufflers to the analyzer is already quite restricted, the analyzer is the sole principal with access privileges to these records. In contrast, in our attack model, the analyzer's output is considered public (although in practice, it may be subject to strict access controls).

For final protection of users' privacy, the analyzer output may make use of the techniques of differentially-private release. This may be achieved by custom analysis mechanisms such as those in PINQ [52] and FLEX [38], or by systems such as Airavat [63], GUPT [53], or PSI [31]. These ready-made systems carefully implement a variety of differentially private mechanisms packaged for use by an analyst without specialized expertise.

However, it may be superfluous for the analyzer to produce differentially-private output. In many cases, shuffling anonymity and encoding fragmentation may suffice for privacy. In other cases, differential privacy may already be guaranteed due to random noise at encoding or to crowd thresholding at shuffling. In such cases, the database of user data is easier to work with (e.g., using SQL or NoSQL tools), as it requires no special protection.

For a protected database, intended to remain secret, exceptional, break-the-glass use cases may still be supported, even though differentially-private release is required to make its analysis output public. Such exceptional access may be compatible with trust assumptions, and even expected by users, and may greatly improve utility e.g., by allowing the analyzer to debug the database and remove corrupt entries.

### 3.5 Privacy Guarantees

Differential privacy [26,27] is a rigorous and robust notion of database privacy. As described earlier, in §2.2, it guarantees that the distribution of analysis outputs is insensitive to the presence or absence of a single record in the input dataset.

Differential privacy provides the following beneficial properties, which may be established at three points with ESA.

**Robustness to auxiliary information** Differential privacy guarantees are independent of observers' prior knowledge, such as (possibly partial) knowledge about other records in the database. In fact, even if all clients but one collude with attackers, the remaining client retains its privacy.

**Preservation under post-processing** Once differential privacy is imposed on analysis output, it cannot be undone. This means that all further processing (including joint computations over outputs of other analysis) do not weaken the guarantees of differential privacy.

**Composability and graceful degradation** Differential privacy guarantees degrade gracefully, in an easily-understood manner. At the encoder, if a single user contributes their data in multiple, apparently-unrelated reports (e.g., because the user owns multiple devices), then that user's privacy only degrades linearly. At the analyzer, the privacy loss is even less—sublinear, in terms of $(\varepsilon, \delta)$ differential privacy—if a sensitive database is analyzed multiple times, with differentially-private release of the output.

Two principal approaches to practical deployment of differential privacy are *local* (see earlier discussion in §2.2) and *centralized*. Centralized differential privacy is typically achieved by adding carefully calibrated noise to the mechanism's outputs. In contrast with local differential privacy, the noise may be smaller than the sampling error (and thus nearly imperceptible) but this requires trusting the analyzer with secure execution of a differentially private mechanism.

The ESA architecture offers a framework for building flexible, private-analytics pipelines. Given an intended analysis problem, a privacy engineer can plug in specific privacy tools at each stage of an ESA pipeline, some of which we have implemented in PROCHLO, and achieve the desired end-to-end privacy guarantees by composing together the properties of the individual stages. We elaborate on such privacy tools for each stage of the ESA pipeline, and describe some illustrative scenarios, below.

***Encoder: Limited Fragments and Randomized Response***

Encoding will limit the reported data to remove superfluous and identifying aspects. However, even when the data itself may be uniquely identifying (e.g., images or free-form text), encoding can provide privacy by fragmenting the data into multiple, anonymous reports. For small enough fragments (e.g., pixel, patch, character, or n-gram), the data may be inherently non-identifying, or may be declared so, by fiat.

Encoders may utilize local differential privacy mechanisms, like RAPPOR [7,28,70,74]. However, for fragments, or other small, known data domains, users may simply probabilistically report random values instead of true ones—a textbook form of randomized response [75]. Finally, for domains with hard-to-guess data, secret-share encoding can protect the secrecy of any truly unique reports (see §4.2).

***Shuffler: Anonymity and Randomized Thresholding***

The shuffler's batching and anonymity guarantees can greatly improve privacy, e.g., by preventing traffic analysis as well as thwarting longitudinal analysis (cf. [70]).

In addition, by using the crowd ID of each report, the shuffler guarantees a minimum cardinality of peers (i.e., a crowd), for reports to blend into [10, 33, 45]. Crowd IDs may be based on user's attributes (e.g., their preferred language) or some aspect of the report's payload (e.g., a hash value of its data). If privacy concerns are raised by revealing crowd IDs to the shuffler, then they can be addressed by using two shufflers and cryptographic crowd IDs blinding, as discussed in §4.3. Even naïve cardinality thresholding—forwarding when $T$ reports share a crowd ID—will improve privacy (provably so, against the analyzer, with the secret sharing of §4.2). However thresholding must done carefully to avoid the $k$-anonymity pitfalls [22,32,33,48].

With randomized thresholding, the shuffler forwards only crowd IDs appearing more than $T + noise$ times, for $noise$ independently sampled from the normal distribution $\mathcal{N}(0, \sigma^2)$, at an application-specific $\sigma$. Subsuming this, shufflers can drop $d$ items from each bucket of crowd IDs, for $d$ sampled from a rounded normal distribution $\lfloor \mathcal{N}(D, \sigma^2) \rceil$ truncated at 0. With near certainty, the shuffler will forward reports whose crowd ID cardinality is much higher than $T$.

The first, similar to the privacy test in Bindschaedler et al. [10], achieves differential privacy for the crowd ID *set*. The second provides improved privacy for the crowd ID counts (at the small cost of introducing a slight systematic bias of dropping $D$ items on average for small $D$). In combination, these two steps address the partitioning concerns raised in §2.2, and allow the shuffler to establish strong differential-privacy guarantees for each crowd.

***Analyzer: Decryption and Differentially Private Release***

Differentially private mechanisms at all stages of the ESA pipeline can be deployed independently of each other; their guarantees will be complementary.

The analyzer, which has the most complete view of the reported data, may choose from a wide variety of differentially private mechanisms and data processing systems [27,31,38, 53,63]. While these analysis mechanisms will typically assume direct access to the database to be analyzed, they do not require the provenance of the database records—fortunately, since they are elided by the shuffler.

For an illustration of flexibility of the ESA architecture, consider two sharply different scenarios. In the first, users report easily-guessable items from a small, known set (e.g., common software settings). While the data in these reports may be of low-sensitivity, and not particularly revealing, it may have privacy implications (e.g., allowing tracking of users). Those concerns may be addressed by the ESA encoder fragmenting users' data into multiple reports, and by

the ESA shuffler ensuring those reports are independent, and anonymous, and cannot be linked together by the analyzer.

For a second example of ESA flexibility, consider when users report sensitive data from a virtually unbounded domain (e.g., downloaded binaries). By using the secret-share ESA encoding, or randomized thresholding at the ESA shuffler—or both—only commonplace, frequent-enough data items may reach the analyzer's database—and the set and histogram of those items will be differentially private. To add further guarantees, and improve long-term privacy, the analyzer can easily apply differentially-private release to their results, before making them public.

# 4. PROCHLO Implementation and Hardening

Assuming separation of trust among ESA steps, each adds more privacy to the pipeline. Even in the worst of our threat models, the encoder's privacy guarantees still hold (§3.1). In this section, we study how ESA can be hardened further, to make our worst threat models less likely. Specifically, in §4.1, we study how trustworthy hardware can strengthen the guarantees of ESA, in particular its shuffler, allowing it to be hosted by the analysis operator. In §4.2 we describe how a novel application of secret-sharing cryptography to the ESA encoder increases our privacy assurances both separately and in conjunction with the shuffler. Then, in §4.3, we explore how a form of cryptographic blinding allows the shuffler to be distributed across distinct parties, to further protect sensitive crowd IDs without compromising functionality.

## 4.1 Higher Assurance by using Trustworthy Hardware

Trustworthy hardware has been an attractive solution, especially after the recent introduction of Intel's SGX, as a mechanism for guaranteeing correct and confidential execution of customer workloads on cloud-provider infrastructure. In particular, for the types of data center analytics that would be performed by the ESA analyzer, there have been several proposals in the recent literature [20, 57, 67].

There has been relatively less proposed for protecting the client side, e.g., the encoders for ESA. Abadi articulated the vision of using secure computing elements as client-local trusted third parties to protect user privacy [1]. We have also argued that trusted hardware can *validate* the user's data, to protect privacy and analysis integrity [47].

In this paper we focus on using trustworthy hardware to harden the ESA Shuffler. In practical terms, trustworthy hardware enables the collocation of the shuffler at the same organization hosting the analyzer, largely eliminating the need for a distinct trusted third party; even though the same organization operates the machinery on which the shuffler executes, trustworthy hardware still prevents it from breaking the shuffler's guarantees (§3.3). This flexibility comes at a cost: since trustworthy hardware imposes many hard constraints, e.g. limiting the size of secure, private memory and allowing only indirect access to systems' input/output features. In this section, we describe how PROCHLO implements the ESA shuffler using Intel's SGX, which in current hardware realizations provides only 92 MB of private memory and no I/O support [36].

### 4.1.1 SGX Attestation for Networked Shuffler Services

To avoid clients having to blindly trust the ESA shuffler over the network, PROCHLO employs SGX to assert the legitimacy of the shuffler and its associated public key. Thereby, per our attack models (§3.1), that key's cryptographic statements can give clients greater assurance that their encoded messages are received by the correct, proper shuffler, without having to fully trust the organization hosting the shuffling service, or its employees.

Specifically, upon startup, the shuffler generates a public/private key pair and places the public key in a Quote operation, which effectively attests that "An SGX enclave running code $X$ published public key $PK_{shuffler}$." The shuffler's hosting organization cannot tamper with this attestation, but can make it public over the network. Having fetched such an attestation, clients verify that it (a) represents code $X$ for a known, trusted shuffler, and (b) represents a certificate chain from Intel to a legitimate SGX CPU. Now the client can derive an ephemeral $K_{shuffler}$ encryption key for every data item it sends to the particular shuffler.

The shuffler must create a new key pair every time it restarts, to avoid state-replay attacks (e.g., with a previously attested but compromised key pair). So, unlike a shuffler running at a trusted third party, an SGX-based shuffler will have shorter-lived public keys—say on the order of hours. Certificate transparency mechanisms may also be used to prevent certificate replay by malicious operators [42, 49].

### 4.1.2 Oblivious Shuffling within SGX Enclaves

*Oblivious shuffling* algorithms produce random permutations of large data arrays via a sequence of public operations on data batches—with each batch processed in secret, in small, private memory—such that no information about the permutation can be gleaned by observing those operations. Due to the limited private memory of trustworthy hardware, random permutations of large datasets must necessarily use oblivious shuffling, with the bulk of the data residing encrypted, in untrusted external memory or disk. Therefore, oblivious shuffling has received significant attention as interest in trustworthy hardware has rekindled [20, 23, 57, 78].

A primitive shuffling operation consists of reading a (small) fraction of records encrypted with some key $Key1$ into private memory, randomly shuffling those records in private memory, and then writing them back out, re-encrypted with some other key $Key2$ (see Figure 2). As long as $Key1$ and/or $Key2$ are unknown to the outside observer, the main observable signal is the sequence of primitive shuffling operations on encrypted-record subsets.

To be secure, the shuffler must apply enough primitive shuffling operations to its input data array so that an ob-

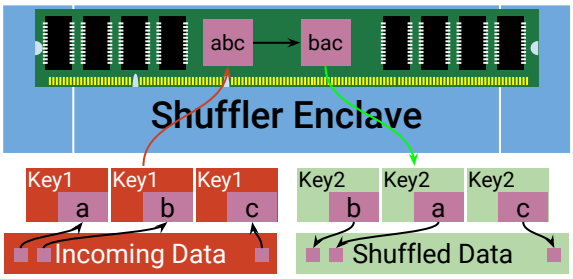


Figure 2: Primitive operation of an oblivious shuffler.

server of primitive shuffling operations gains no advantage in recovering the resulting order compared to guessing at random; this induces the *security parameter*, $\epsilon$, which is defined as the total variation distance between the distribution of shuffled items and the uniform distribution. Typical values range from the bare minimum of $\epsilon = 1/N$ to a cryptographically secure $\epsilon = 2^{-128}$.

To be efficient, the shuffler must perform as few primitive shuffling operations as possible, since each such operation imposes memory overhead (to bring a subset into private memory, and write it back out again), cryptographic overhead (to decrypt with $Key1$ and re-encrypt with $Key2$), and possibly network overhead, if multiple computers are involved in running primitive shuffling operations.

#### 4.1.3 State of the Art in Oblivious Shuffling

We attempted to find existing oblivious-shuffling mechanisms that produced cryptographically-secure random permutations with sufficient scalability and efficiency.

As described below, we found no SGX oblivious shuffling mechanisms adequate to our purposes, leading to the design and implementation of our own algorithm, the Stash Shuffle (§4.1.4). In terms of scalability, our RAPPOR experience shows that we must handle tens to hundreds of millions of items, with at least 64 bytes of data and an 8-byte integer crowd ID, each, which in PROCHLO corresponds to a 318-byte doubly-encrypted record. In terms of efficiency, our driving constraint came from the relative scarcity of SGX-enabled CPUs in data centers, resulting from SGX's current deployment only as a per-socket (not per-core) feature of client-class CPU realizations. We defined our metric for efficiency in terms of the total amount of SGX-processed data, relative to the size of the input dataset; thus, at 2× efficiency SGX would read, decrypt, re-encrypt, and write out to untrusted memory each input data item two times.

Oblivious sorting is one way to produce an oblivious shuffle: associate a sufficiently-large random identifier (say 64 bits) with every item, by taking a keyed hash of the item's contents, and then use an oblivious sorting algorithm to sort by that identifier. Then the resulting shuffled (sorted) order of items will be as random as the choice of random identifier. Oblivious sorting algorithms are sorting algorithms that choose the items to compare and possibly swap (inside private memory) in a data-independent fashion.

Batcher's sort is one such algorithm [8]. Its primitive operation reads two buckets of $b$ consecutive items from an array of $N$ items, sorts them by a keyed item hash, and writes them back to their original array position. While not restricting dataset size, Batcher's sort requires $N/2b \times (\log_2 N/b)^2$ such private sorting operations. With SGX, $b$ can be at most 152 thousand 318-byte records. Thus, to apply Batcher's sort to 10 million records (a total of 3.1 GBytes), the data processed will be 49× the dataset size (155 GBytes); correspondingly, for 100 million records (31 GBytes), the overhead would be 100× (3.1 TBytes). To complete this processing within a reasonable time, at least each day, many SGX machines would be required to parallelize the rounds of operations: 33 machines for 10 million records, and 330 for 100 million—a very substantial commitment of resources for these moderately small datasets.

ColumnSort is a more efficient, data-independent sorting algorithm, improving on Batcher's sort's $\mathcal{O}((\log_2 N)^2)$ rounds and sorting datasets in just exactly 8 rounds [44]. Opaque uses SGX-based ColumnSort for private-data analytics [78]. Unfortunately, any system based on ColumnSort has a maximum problem size that is induced by the per-machine private-memory limit [14]. Thus, while ColumnSort's overhead is only 8× the dataset size, it can at most sort 118 million 318-byte records.

Sorting is a brute-force way to shuffle: instead of producing *any* unpredictable data permutation, it picks exactly *one* unpredictable permutation and then sorts the data accordingly. A bit further away from sorting lies the Melbourne Shuffle algorithm [58]. It is fast and parallelizable, and has been applied to privacy-data analytics in the cloud [57]. Instead of picking random identifiers for items and then sorting them obliviously, the Melbourne shuffle picks a random permutation, and then obliviously rearranges data to that ordering. This rearrangement uses data-independent manipulations of the data array, without full sorting, which reduces overhead. Unfortunately, this algorithm scales poorly, since it requires access to the entire permuted ordering in private memory. For SGX, this means that the Melbourne Shuffle can handle only a few dozen million items, at most, even if we ignore storage space for actual data, computation, etc.

Finally, instead of sorting, cascade-mix networks have been proposed for oblivious shuffling (e.g., M2R [23]). With SGX, such networks split the input data across SGX enclaves, partitioned to fit in SGX private memory, and redistribute the data after random shuffling in each enclave. A "cascade" of such mixing rounds can achieve any permutation, obliviously. Unfortunately, for a safe security parameter $\epsilon = 2^{-64}$, a significant number of rounds is required [40]. For our running example, the overhead is 114× for 10 million 318-byte records, and 87× for 100 million records.

### 4.1.4 The Stash Shuffle Algorithm

Since existing oblivious shuffling algorithms did not provide the necessary scalability, efficiency, or randomness of the permutations, we designed and implemented the *Stash Shuffle*. Its security analysis is in a separate report [50].

Our algorithm is inspired by the Melbourne Shuffle, but does not store the permutation in in private memory. As with other oblivious shuffle algorithms, it reads $N$ encrypted items from untrusted memory, manipulates them in buckets small enough to fit in private memory, and writes them out as $N$ randomly-shuffled items, possibly in multiple rounds. In the case of ESA, the input consists of doubly-encrypted data items coming from encoders, while the output consists of the inner encrypted data item only (without crowd IDs or the outer layer of encryption).

**Algorithm 1** The Stash Shuffle algorithm.

```
1: procedure StashShuffle(Untrusted arrays in, out, mid)
2:     stash ← φ
3:     for j ← 0, B − 1 do
4:         DistributeBucket(stash, j, in, mid)
5:     DrainStash(stash, B, mid)
6:     FAIL on ¬stash.Empty()
7:     Compress(mid, out)
```

Algorithm 1, the Stash Shuffle, considers input (*in*) and output (*out*) items in $B$ sequential buckets, each holding at most $D \triangleq \lceil N/B \rceil$ items, sized to fit in private memory. At a high level, the algorithm first chooses a random output bucket for each input item, and then randomly shuffles each output bucket. It does that in two phases. During the *Distribution Phase* (lines 2–6), it reads in one input bucket at a time, splits it across output buckets, and stores the split-up but as yet unshuffled, re-encrypted items in an intermediate array (*mid*) in untrusted memory. During the *Compression Phase* (line 7), it reads the intermediate array of encrypted items one bucket at a time, shuffles each bucket, and stores it fully-shuffled in the output array.

The algorithm gets its name from the *stash*, a private structure, whose purpose is to reconcile obliviousness with the variability in item counts distributed across the output buckets. This variability (an inherent result of balls-and-bins properties) must be hidden from external observers, and not reflected in non-private memory. For this, the algorithm caps the number traveling from an input bucket to an output bucket at $C \triangleq D/B + \alpha\sqrt{D/B}$ for a small constant $\alpha$. If any input bucket distributes more than $C$ items to an output bucket, overflow items are instead stored in a stash—of size $S$—where they queue, waiting to be drained into the chosen output bucket during processing of later input buckets.

Figure 3 illustrates the distribution phase for the second of four buckets, where $B = 4$, $D = 6$, and $C = 2$. In Step 1, the bucket is read into private memory and decrypted. It is shuffled into the 4 destination buckets in Step 2, depositing three items in the first target bucket, one in the second, none in the third, and two in the last. Since the stash already

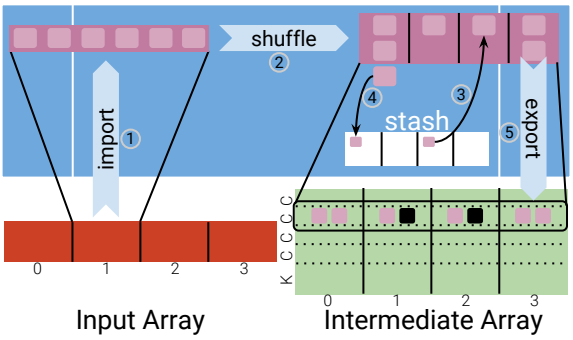


Figure 3: Distribution phase.

held one item for the third target bucket, that item is moved into the third destination bucket, in Step 3. The first destination bucket got three items but it can only deposit two, so the third item is stashed away in Step 4. Finally, the items are re-encrypted and deposited out into the intermediate array, filling in any empty slots with encrypted dummy items to avoid leaking to the outside how items were distributed (shown black in the figure), in Step 5.

**Algorithm 2** Distribute one input bucket.

```
1: procedure DistributeBucket(stash, b, Untrusted arrays in, mid)
2:     output ← φ
3:     targets ← ShuffleToBuckets(B, D)
4:     for j ← 0, B − 1 do
5:         while ¬output[j].Full() ∧ ¬stash[j].Empty() do
6:             output[j].Push(stash[j].Pop())
7:     for i ← 0, D − 1 do
8:         item ← Decrypt(in[DataIdx(b, i)])
9:         if ¬output[targets[i]].Full() then
10:            output[targets[i]].Push(item)
11:        else
12:            if ¬stash.Full() then
13:                stash[targets[i]].Push(item)
14:            else
15:                FAIL
16:    for j ← 0, B − 1 do
17:        while ¬output[j].Full() do
18:            output[j].Push(dummy)
19:        for i ← 0, C − 1 do
20:            mid[MidIdx(j, i)] ← Encrypt(output[j][i])
```

Algorithm 2 describes the distribution in more detail, implementing the same logic, but reducing data copies. ShuffleToBuckets randomly shuffles the $D$ items of the input bucket, and $B-1$ bucket separators. The shuffle determines which item will fall into which target bucket, stored in *targets* (line 3). Then, for every output bucket, as long as there is still room in the maximum $C$ items to output, and there are stashed away items, the output takes items from the stash (lines 4–6). Then the input bucket items are read in from the outside input array, decrypted, and deposited either in the output (if there is still room in the quota $C$ of the target bucket), or in the stash (lines 7–15). Finally, if some output chunks are still not up to the $C$ quota, they are filled with dummy items, encrypted and written out into the intermediate array (lines 16–20). Note that the stash may end up with

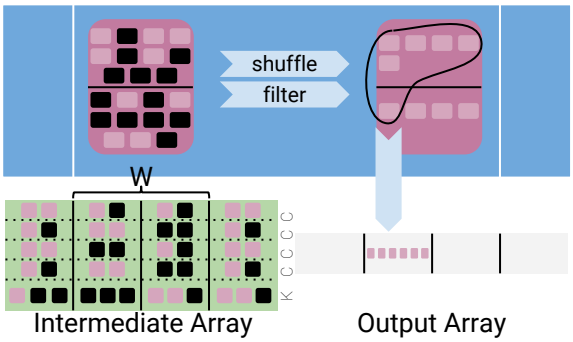


Figure 4: Compression phase.

items left over after all input buckets have been processed, so we drain those items (padding with dummies), filling $K$ extra items per output bucket at the end of the distribution phase (line 5 of Algorithm 1, which is similar to distributing a bucket, except there is no input bucket to distribute). $K$ is set to $S/B$, that is, the size of the stash divided by the number of buckets.

**Algorithm 3** Compress intermediate items.
($L \triangleq \min(W, B)$ is the effective window size, defined to account for pathological cases where $W > B$.)

```
1: procedure COMPRESS(Untrusted arrays mid, out)
2:     for b ← 0, L − 1 do
3:         IMPORTINTERMEDIATE(b, mid)
4:     for b ← L, B − 1 do
5:         DRAINQUEUE(b − L, mid, out)
6:         IMPORTINTERMEDIATE(b, mid)
7:     for b ← B − L, B − 1 do
8:         DRAINQUEUE(b, mid)
```

Algorithm 3 and Figure 4 show the compression phase. In this phase, the intermediate items deposited by the distribution phase must be shuffled, and dummy items must be filtered out. To do this, without revealing information about the distribution of (real) items in output buckets, the phase proceeds in a sliding window of $W$ buckets of intermediate items. The window size $W$ is meant to absorb the elasticity of real item counts in each intermediate output bucket due to the Binomial distribution. See

**Algorithm 4** Import an intermediate bucket.

```
1: procedure IMPORTINTERMEDIATE(b, Untrusted array mid)
2:     bucket ← mid[MidIdx(b, 0..C ∗ B + K − 1)]
3:     Shuffle(bucket)
4:     for i ← 0, C ∗ B + K − 1 do
5:         item ← Decrypt(bucket[i])
6:         if ¬item.dummy then
7:             queue.Enqueue(item)
```

As Algorithm 4 shows, an intermediate bucket is loaded into private memory ($C$ items per input bucket, plus another $K$ items for the final stash drain) in line 2, and shuffled in line 3. Then intermediate items are decrypted, throwing away dummies, and enqueued for export, $D$ items at a time,

| $N$ | $B$ | $C$ | $W$ | $S$ | **log($\epsilon$)** | **Overhead** |
|---|---|---|---|---|---|---|
| 10M | 1,000 | 25 | 4 | 40,000 | -80.1 | 3.50× |
| 50M | 2,000 | 30 | 4 | 86,000 | -81.8 | 3.40× |
| 100M | 3,000 | 30 | 4 | 117,000 | -81.9 | 3.70× |
| 200M | 4,400 | 24 | 4 | 170,000 | -64.5 | 3.32× |

Table 1: Stash Shuffle parameter scenarios, their security, and relative processing overheads, assuming 318-byte encrypted items (64 data bytes and 8-byte crowd IDs).

into the output array in untrusted memory. In Figure 4, continuing in the same example, two intermediate buckets are in private memory at steady state (i.e., $W = 2$), shuffled, and streamed in their final order to the output array, $D = 6$ items per output bucket, after filtering out dummies.

Even with the stash, the algorithm may fail. During distribution, the stash may fill up or fail to fully drain in the end, and during compression the queue may fill up when importing. Upon failure, the algorithm aborts and starts anew. Since intermediate items are encrypted with an ephemeral symmetric key, failed attempts leak no information about the eventual successfully-shuffled order. However, failures are possible because some permutations are infeasible to the algorithm for given parameters. For example, the identity permutation is infeasible unless $C \geq D$, because it would require all $D$ items of every input bucket $b$ to land in the same $C$-sized chunk $output[b]$ in Algorithm 2, line 10. The security analysis for a choice of $N$, $B$, $C$, $S$, and $W$ determines the security parameter $\epsilon$, which indirectly reflects the fraction of infeasible permutations [50]. Table 1 shows a few choices for our 318-byte encrypted items, the resulting security parameter, and the processing overhead; in the Stash shuffle's case, we process $N$ data items and $B^2C + S$ intermediate items. Both scaling and efficiency are significantly improved compared to prior work (§4.1.3).

The table shows problem sizes up to 200 million items, at which point the algorithm as we defined it above reaches its scalability limits for SGX (for 318-byte items and safe security parameters). To scale to larger problems, the algorithm can be modified to spill internal state to untrusted memory, increasing overall processing overhead to roughly double, or can be run twice in succession with smaller security parameters, which has the effect of boosting the overall security of shuffling (this is a standard technique, presented as a default choice for the Melbourne shuffle [58]).

Oblivious shuffling benefits from parallelism. Previous solutions, including the Melbourne shuffle, Batcher's sort, Column sort, and cascade-mix networks consist of 4, $(\log_2 N/b)^2$, 8, and $\mathcal{O}(\log B)$, respectively, embarrassingly parallel rounds, run sequentially. The Stash Shuffle Distribution phase parallelizes well, by replicating the stash to each worker, and splitting a somewhat larger $K$ among workers. However, its Compression phase is largely sequential. Thankfully, the sequential Compression phase is inconsequential in practice, since it involves only symmetric cryp-

tography, whereas the Distribution phase performs public-key operations to derive the shared secret key $K_{shuffler}$ with which to decrypt the outer layer of nested encryption (§5.1).

The above description is aimed at giving the reader some intuition. In our implementation, we took care to avoid memory copies in private memory, because they incur a cryptographic cost due to SGX's Memory Encryption Engine. For example, we use only statically allocated private arrays organized as queues, stacks, and linked lists, rather than standard containers employing dynamic memory allocation; we shuffle index arrays rather than arrays of possibly large data items; and we overlay the distribution data structures in the same enclave memory as the compression data structures. We also took care to minimize side channels [12, 43, 76]; for example, we ensure that control paths involving real and dummy items are the same, and write the same private memory pages. In terms of performance, our implementation is untuned. For example, to avoid issuing system OCALLs (i.e., calls out of the enclave into untrusted space to invoke system functionality), we memory-map our three big arrays (input batch, intermediate array, output batch) and let virtual memory handle reading and writing from secondary storage. Given the predictable access patterns of the stash shuffler (sequential read of the input batch, sequential stride write of the intermediate array, sequential read of the intermediate array, sequential write of the output batch), we gain a benefit from intelligent prefetching using `madvise` on the memory-mapped ranges, to reduce the cost of I/O.

#### 4.1.5 Crowd Cardinality Thresholding inside SGX

Thresholding from inside an enclave with limited private memory requires some care. It must not reveal to the hosting organization the distribution of data items over crowd IDs, or the values of crowd IDs themselves; we chose to allow the hosting organization to see the selectivity of the thresholding operation (i.e., the number of data records that survived filtering), since the analyzer would have access to this information anyway.

To perform thresholding, the shuffler must perform two tasks obliviously: counting data items with the same crowd ID, and filtering out those with crowd IDs with counts below the threshold. In the simpler case, the crowd ID domain—that is, all distinct crowd ID values—is small enough that the shuffler can fit one counter per value in private memory; for SGX enclaves, those are crowd ID domains of about 20 million distinct values. In that case, counting requires one pass over the entire batch, updating counters in private memory, and another pass to filter out (e.g., zero out) those data items with crowd IDs with counts below threshold. If oblivious shuffling has already taken place (§4.1.2), these two scans of the shuffled data items reveal no information other than the global selectivity of the thresholding operation. All of the problems to which we have applied ESA have small-enough crowd ID domains to be countable inside private memory and can, therefore, be thresholded in this manner.

Should a problem arise for which the crowd ID domain is too big to fit inside private memory, a more expensive approach is required. For example, the batch can be obliviously sorted (e.g., via Batcher's sort) by crowd ID, and then scanned one more time, counting the data items in a single crowd, associating a running count with each item (via re-encryption). In a final pass (in the opposite scanning direction), the count of each crowd ID is found in the first data item scanned backwards, so all items by that crowd ID can be filtered out. Since this approach requires oblivious sorting anyway, it can be combined with the shuffling operation itself, obviating the need for something more efficient like the Stash Shuffler; in fact, this is a specialized form of more-general oblivious relational operators, such as those proposed in Opaque [78]. More efficient approaches employing approximate counting (e.g., counting sketches) may lead to faster oblivious thresholding for large crowd ID domains. Nevertheless, we have yet to encounter such large crowd ID domains in practice.

### 4.2 Encoding Using Secret-Sharing Cryptography

The ESA architecture enables applications of novel encoding algorithms to further protect user data. One such encoding scheme, constructed using *secret sharing*, can guarantee data confidentiality for user data drawn from a distribution with unique and hard-to-guess items, such as fingerprints, or random URLs. (We call this secret sharing, instead of *threshold cryptography*, so as to not overload the word "threshold.") This scheme composes well with randomized thresholding at the shuffler (see §3.5). When combined with the blinded crowd IDs discussed next (in §4.3), this scheme provides strong privacy for for both easy-to-guess, limited-domain data, as well hard-to-guess, unique data. These benefits are demonstrated for a concrete example in §5.2.

First, recall the basic idea behind Shamir secret sharing [68]: $t$-secret sharing splits a secret $s$ in a field $\mathbb{F}$ into arbitrary shares $s_1, s_2, \ldots$, so that any $t-1$ or fewer shares statistically hide all information about $s$. Given any $t$ shares one can recover $s$ completely, by choosing a $(t-1)$-degree polynomial $P \in \mathbb{F}[X]$ with random coefficients subject to $P(0) = s$. A secret share of $s$ is the tuple $(x, P(x))$ for randomly chosen non-zero $x \in \mathbb{F}$. Given $t$ points on the curve, Lagrange interpolation recovers $P$ and $s = P(0)$.

From the above, a novel *$t$-secret share encoding* of an arbitrary string $m$ with parameter $t$ is a pair $(c, aux)$ constructed as follows. Ciphertext $c$ is a deterministic encryption of the message under a *message-derived key* [3,9] $k_m = H(m)$ and $aux$ is a (randomized) $t$-secret share of $k_m$. It is crucial to note that these secret shares can be computed independently by users, which enables its direct application as an encoding scheme in the ESA architecture.

Correctness of this encoding is easy to see: with overwhelming probability, given $t$ independent shares corresponding to the same ciphertext $c$, one can decrypt it to

recover $m$ by first using secret shares $aux_1, \ldots, aux_t$ to derive $k_m$ and then using $k_m$ to decrypt $c$.

This encoding protects the secrecy of $m$ with fewer instances that $t$. With at most $t-1$ independent shares corresponding to the same ciphertext $c$, the values $\langle aux_1, \ldots, aux_{t-1}, c\rangle$ reveal no information about $m$ that cannot anyway be guessed about $m$ by an adversary *a priori*. The security analysis follows by combining statistical security of Shamir secret sharing and by that of a message-derived key [3, 9]. As noted before, if $m$ comes from a distribution that is hard to guess, no information about $m$ is leaked until a minimum of $t$ encodings is collected.

### 4.3 Blinded Crowd IDs for Shuffler Thresholding

Using novel public-key cryptography, we enable thresholding on *private* crowd IDs in PROCHLO, when a crowd ID might itself be sensitive. Specifically, we have designed and implemented a split shuffler, consisting of two non-colluding parties, which together shuffle and threshold a dataset without accessing a crowd ID in the clear. Rather than sending the crowd ID encrypted to the shuffler's private key, the encoder hashes the crowd ID to an element of a group of a prime order $p$ as $\mu = H(\text{crowd ID})$. In our implementation the group is the elliptic curve NIST P-256 [55] and in this presentation, we use multiplicative notation for clarity. The encoder computes an El Gamal encryption $(g^r, h^r \cdot \mu)$, where $(g, h)$ is Shuffler 2's public key and $r$ is a random value in $\mathbb{Z}_p$. Shuffler 1 *blinds* the tuple with a secret $\alpha \in \mathbb{Z}_p$ to compute $(g^{r\alpha}, (h^r \cdot \mu)^\alpha)$, and then batches, shuffles, and forwards the blinded items (with the usual accompanying data) to Shuffler 2. Shuffler 2 uses its private key, i.e., the secret $x$ such that $h = g^x$, on input $(u, v)$ to compute $v/u^x$ and recovers $\mu^\alpha = H(\text{crowd ID})^\alpha$.

At the cost of three extra group exponentiations, Shuffler 2 now works with crowd IDs that are hashed and raised to a *secret* power $\alpha$. Critically, blinding preserves the equality relation, which allows comparison and counting.

Note that hashing alone would not have protected the crowd ID from a dictionary attack. With blinded crowd IDs, Shuffler 1 cannot mount such an attack, since it does not possess Shuffler 2's private key, and Shuffler 2 cannot mount such an attack, because it does not posses Shuffler 1's secret $\alpha$. As long as Shufflers 1 and 2 do not collude, this mechanism enables them to (jointly) threshold on crowd IDs without either party having access to it in the clear.

This form of private thresholding is useful in several scenarios: (a) when crowd IDs involve ZIP code or other more personal and identifying information that an adversary might be able to guess well; (b) when collecting data as in §5.5 where low-frequency data points such as obscure movie ratings do little to add utility but enable easy de-anonymization; and (c) when secret-share encoding is applied to data that might be easy to guess. §5.2 presents a comprehensive use case combining blinded crowd IDs with secret-share encoding for analyses with sensitive crowd IDs.

| $N$ | Distribution | Compression | Total | SGX Mem |
|---|---|---|---|---|
| 10M | 713 s | 26 s | 738 s | 22 MB |
| 50M | 3,581 s | 168 s | 1.0 h | 52 MB |
| 100M | 7,172 s | 349 s | 2.1 h | 78 MB |
| 200M | 14,267 s | 620 s | 4.1 h | 69 MB |

Table 2: Stash Shuffle execution of the scenarios in Table 1. Rows show input size; columns show per-phase and total execution time and the maximum private SGX memory used.

### 4.4 Implementation

The PROCHLO framework is implemented in 1100 lines of C++ using OpenSSL and gRPC [35, 73], with another 1600 lines of C++ and OpenSSL code for cryptographic hardening. The Stash Shuffle is implemented in 2300 lines of C++ code, using SGX enclaves in "pre-release" mode and a combination of OpenSSL and the Linux SGX SDK crypto libraries [37]. An open-source implementation is gradually released at `https://github.com/google/prochlo`. Another implementation of the ESA architecture, eventually intended to include all the features of PROCHLO, is at `https://fuchsia.googlesource.com/cobalt`.

## 5. Evaluation

We study four data pipelines that we have ported to PROCHLO. In each case, we motivate a realistic scenario and describe encoders, shufflers, and analyzers tailored to demonstrate the flexibility of ESA. We seek to understand how the utility of each analysis is affected by introducing ESA privacy. We use consistent parameters for the thresholding and noisy loss applied by the shufflers (§3.5). The thresholds are set to 20. Before thresholding is applied, the shufflers drop $d$ items from each bucket. The random variable $d$ is sampled from the rounded normal distribution $\lfloor \mathcal{N}(D, \sigma^2) \rceil$, where $D = 10$ and $\sigma = 2$. This guarantees $(2.25, 10^{-6})$-approximate differential privacy for the multi-set of crowd IDs that the analyzers receive. These settings are at least as strong as in recent industrial deployments and in literature [4, 28, 70]. We first evaluate our SGX-based Stash Shuffler; the four case studies that follow use non-oblivious shufflers.

### 5.1 Stash Shuffle

We measured the Stash Shuffler with datasets of 318-byte items (corresponding to 64 bytes of data and 8 bytes of crowd ID), on an SGX-enabled 4-core Intel i7-6700 processor, 32 GB of RAM, and a Samsung 850 1-TB SSD. The nested cryptography uses authenticated encryption, with NIST P-256 asymmetric key pairs used to derive AES-128 GCM symmetric keys. Table 2, shows our measurement results, from single-threaded runs that form a basis for assessing scalability. Although the two phases process roughly the same amount of data, Distribution is far costlier, because of public-key cryptography, but parallelizes well. Run with 10 SGX workers, approximate execution times would be 1.6, 8.8, 17.8, and 34.1 minutes, respectively.

### 5.2 Vocab: Empirical Long-tail Distributions

We consider a corpus of three billion words that is representative of English-speaking on-line discussion boards. Characteristically, the distribution follows the power-law (Zipfian) distribution with a heavy head and a long tail, which poses a challenge for statistical techniques such as randomized response. To demonstrate PROCHLO's utility into recovering a stronger signal further into the tail of the distribution, we performed the following four experiments to privately learn word frequencies on samples of size 10K, 100K, 1M, and 10M drawn from the same distribution. In each experiment, we measured the number of *unique* words (which can be thought of as unique candidate URLs or apps in other applications) we could recover through our analysis.

In experiment Crowd, clients send *unencoded* words along with a hash of the word as the crowd ID to a single shuffler. Against the analyzer, this hides all words that occur infrequently and allows decoding of words whose frequency is above the threshold. However, a malicious shuffler may mount a dictionary attack on the words' hashes, and there is no privacy against the shuffler and analyzer colluding.

Experiment Secret-Crowd builds on Crowd, but clients encode their reported words using one-out-of-$t$ secret sharing, setting $t$ to be 20, like the shuffler's crowd threshold $T$. At a minimal computational cost to clients (less than 50 $\mu$s per encoding), privacy is significantly improved: uncommon words and strings drawn from hard-to-guess data sources (such as private keys, hash values, love letters, etc.) are private to the analyzer. Alas, the shuffler can mount dictionary attacks and statistical inference on crowd IDs.

Experiment NoCrowd uses the same secret sharing as Secret-Crowd, but uses the same, fixed crowd ID in all client reports. This protects against a malicious shuffler, as it no longer can perform statistical inference or dictionary attacks on crowd ID word hashes. Also, this slightly improves utility by avoiding the small noise added by the shuffler during the thresholding step. However, lacking a crowd to hide in, clients now have less protection against the analyzer: it will now receive reports even of the most uncommonly-used words, and can attempt brute-force attacks on them.

Experiment Blinded-Crowd offers the most compelling privacy story. In addition to secret-share encoding of words, clients use blinded crowd IDs, with two-shuffler randomized thresholding (§4.3). Assuming no collusion, neither the shufflers nor the analyzer can successfully perform attacks on the secret-shared words or the blinded crowd IDs. Even if all parties collude, private data from a hard-to-guess distribution (such as keys and unique long-form text) will still be protected by the secret-share encoding.

For each experiment, we compute a histogram and measure utility based on the number of unique words recovered in the analysis. Finally, we compare PROCHLO with RAPPOR [28] and its variant where collected reports are partitioned by small crowd IDs a few bits long (see the discussion of local differential privacy in §2.2). This translates to between 4 and 256 partitions for the sample sizes in the experiment. The results are summarized in Figure 5.

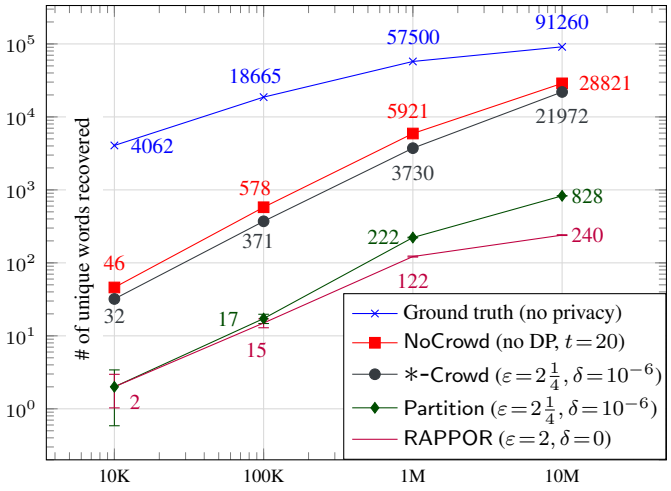


Figure 5: A log-log-graph of the number of unique words recovered (Y-axis) on samples of 10 thousand to 10 million Vocab words (X-axis). Using results in The $*$-Crowd line results from using word hashes as the crowd-IDs, whereas NoCrowd offers less privacy, using a naïve threshold of 20 and no crowds. For comparison, RAPPOR and Partition show how pure local-differential privacy offers far less accuracy and much higher variance (error bars) even when augmented with partitions as described in §2.2.

Several observations are in order. The experiment offering the highest utility is NoCrowd which performs no crowd-based thresholding, but also provides no differential privacy guarantees, unlike the other experiments. Encouragingly, the $*$-Crowd experiments show the utility loss due to noisy thresholding to be very small compared to NoCrowd. Both experiments recover a large fraction of the ground-truth number of unique words computed without any privacy.

The challenge of using randomized response for long-tailed distributions is made evident by the RAPPOR results, whose utility is less than 5% that of our PROCHLO experiments. The Partition results also show that the limitations of local differential privacy cannot be mitigated by following §2.2. and partitioning based on word hashes. For the Vocab dataset, and the studied sample sizes, partitioning improves RAPPOR's utility only by between $1.13\times$ to $3.45\times$, at the cost of relaxing guarantees from 2-differential privacy to $(2.25, 10^{-6})$-differential privacy.

Table 3 gives wall-clock running time for the Vocab experiment across varying problem sizes. Performance was measured on an 8-core 3.5 GHz Intel Xeon E5-1650 processor with 32 GB RAM, with multiple processes communicating locally via gRPC [35]. We note that these numbers demonstrate what we naturally expect in our system design: performance scales linearly with the number of clients and the dominating cost is public-key crypto operations (roughly 3, 6, and 2 operations for each column, respectively).

| # clients | Encoder+Shuffler 1 | | Shuffler 2 |
|---|---|---|---|
| | {Secret-C,NoC,C} | Blinded-C | Blinded-C |
| 10K | 8 s | 15 s | 7 s |
| 100K | 71 s | 153 s | 64 s |
| 1M | 713 s | 0.4 h | 643 s |
| 10M | 2.0 h | 4.1 h | 1.8 h |

Table 3: Execution time for the Vocab experimental setup for one shuffler and for two shufflers (with blind thresholding).

| | Geolocation | Notification | Audio |
|---|---|---|---|
| **Naïve Thresh.** | 6,610 | 12,200 | 620 |
| **Granted** | 5,850 | 8,870 | 440 |
| **Denied** | 5,780 | 8,930 | 430 |
| **Dismissed** | 5,860 | 9,465 | 440 |
| **Ignored** | 5,850 | 11,020 | 530 |

Table 4: Number of Web pages recovered using a naïve threshold or, for each user action, a noisy crowd threshold.

### 5.3 Perms: User Actions Regarding Permissions

Consider monitoring new feature adoption in the Chrome Web browser—a rich platform with hundreds of millions of users. For some Chrome features, Web pages are required to ask users for access permissions; users may respond to those requests by accepting, denying, dismissing, or ignoring them. To measure feature adoption and detect Web-page abuse, these requests must somehow be monitored by Chrome developers. PROCHLO provides new means for monitoring with both high utility and strong privacy.

From existing monitoring data, we crafted a dataset for the *Geolocation*, *Notifications*, and *Audio Capture* permissions. The millions of $\langle \text{page}, \text{feature}, \text{action bitmap} \rangle$ tuples in the dataset use bitmap bits for the *Grant*, *Deny*, *Dismiss*, and *Ignore* user actions, since a user sometimes gives multiple responses to a single permission prompt. The entire dataset is privacy-relevant, since it involves the Web pages visited, features used, and actions taken by users.

In our experiments, we performed one simple analysis of this dataset: for each of the $3 \times 4$ feature/user-action combinations, find the set of Web pages that exhibited it at least 100 times. As described in Fanti et al. [30], such multidimensional analysis over long-tail distributions (of Web pages) is a poor fit for local differential privacy methods. Confirming this, we were unable to recover more than a few dozen Web pages, in total, when applying RAPPOR to this dataset.

As shown in Table 5.4, using PROCHLO improves utility of this simple analysis by several orders of magnitude, compared to the above RAPPOR baseline. Also, PROCHLO greatly simplifies software engineering by materializing a database of Web page names, instead of RAPPOR's noisy statistics. Most encouragingly, PROCHLO can offer at least $(\varepsilon = 1.2, \delta = 10^{-7})$-differential privacy, which compares very favorably. This is achieved by the PROCHLO shuffler using a threshold of 100, with Gaussian noise $\sigma = 4$, and crowd IDs based on blinded, secret-shared encryption of $\langle \text{page}, \text{feature} \rangle$. Furthermore, with probability $10^{-4}$, each bitmap bit is flipped in the encoded $\langle \text{page}, \text{feature}, \text{action bitmap} \rangle$ tuples of PROCHLO reports, providing plausible deniability for user actions.

### 5.4 Suggest: Predicting the Next Content Viewed

Next we consider a use case that concerns highly privacy-sensitive input data and existing, hard-to-change analysis software, in the context of YouTube. For this use case, the analysis is custom code for training a multi-layer, fully-connected neural network that predicts videos that users may want to view next, given their recent view history. Content popularity follows a long-tail distribution, and the input data consists of longitudinal video views of different users. The resulting state-of-the-art deep-learning sequence model is based on ordered views of the half-million most popular videos, each with at least tens-of-thousands of views.

Similar sequence prediction models are common, and can be widely useful (e.g., for predicting what data to load into caches based on users' software activity). However, sequence-prediction analysis is inherently at odds with privacy: its input is a longitudinal history of individual users' data—and some parts of that history may be damaging or embarrassing to the user, while other parts may be highly identifying. In particular, for the video view input data, any non-trivial sequence of $n$ views is likely close to unique, as it forms an ordered $n$-tuple over 500,000-item domain.

For this example use case, it is neither possible to modify existing, complex analysis code nor to remove the inherently identifying sequence ordering upon which the analysis relies. Therefore, to protect users' privacy, we implemented a PROCHLO encoding step that fragmented each user's view history into short, disjoint $m$-tuples ($m \ll n$), and relied on PROCHLO shuffling to guarantee that only such tuples could be analyzed—i.e., only anonymous, disassociated very short sequences of views of very popular videos.

This construction is appealingly simple and provides a concrete, intuitive privacy guarantee: for small-enough $m$ (i.e., as $m$ approaches 1), any single $m$-tuple output by the encoder can be identifying or damaging, but not both. Thus, privacy can be well protected by the shuffler preventing associations from being made between any two disjoint tuples.

Fortunately, because recent history is the best predictor of future views, a model trained even with 3-tuples correctly predicts the next view more than 1 out of 8 times, with around 90% of the accuracy of a model trained without privacy. Training each model to convergence on about 200 million longitudinal view histories takes two days on a small cluster with a dozen NVIDIA Tesla K20 GPUs, using TensorFlow [2]. Even more relevant is that this privacy-preserving model has equivalent quality as the best-known YouTube model for this task from about a year ago—when evaluated using an end-to-end metric that models presenting users with multiple suggestions for what to view next.

### 5.5 Flix: Collaborative Filtering

We next consider a prediction task without the strong locality inherent to our good results for next-viewed content in the previous section. This task is to infer users' ratings for content given the *set* of each user's content ratings. Work on this problem was supercharged by the $1M Netflix Prize challenge for improving their proprietary recommender system. Famously, Narayanan and Shmatikov [54] managed to deanonymize users in the challenge's large corpus of training data by exploiting auxiliary data and the linkability of users' movie preferences. Their successful attack led to the cancellation of a followup competition.

We demonstrate that the ESA architecture permits collection of data that simultaneously satisfies dual (and dueling) objectives of privacy and utility for collaborative filtering. As in the previous example, sending a complete vector of movie ratings exposes users to linking attacks. The equally unsatisfying alternative is to guarantee privacy to all users by randomizing their vectors client-side. Since the ratings and the movies are sensitive information, both need to be randomized, effectively destroying data utility.

To enable utility- and privacy-preserving collection of data we identify sufficient statistics that can be assembled from anonymized messages drawn from a small domain. We observe that many of the most relevant analysis methods of collaborative filtering comprise two distinct steps: (i) computing the *covariance matrix* capturing item-to-item interactions, and (ii) processing of that matrix, e.g., for factorization or de-noising. Computation on the sensitive data of users' movie ratings is performed only in the first step, which can be supported by the ESA architecture, as described below.

Let the rating given to item $i$ by user $u$ be $r_{ui}$, the set of items rated by user $u$ be $I(u)$ and the set of users who rated item $i$ be $U(i)$. Towards the goal of evaluating the covariance matrix we compute two intermediate item-by-item matrices $S$ and $A$ defined as $S_{ij} = |U(i) \cap U(j)|$ and $A_{ij} = \sum_{u \in U(i) \cap U(j)} r_{ui} r_{uj}$. An approximation to the covariance matrix is given by $(A_{ij}/S_{ij})$.

We describe how $A$ is computed ($S$ is treated analogously). By pivoting to users, we represent $A$ as follows: $A_{ij} = \sum_u \sum_{i,j \in I(u)} r_{ui} r_{uj}$. Thus, it is sufficient for each user to send its contribution to $A$ that consists of all $(i, r_{ui}, j, r_{uj})$ four-tuples where $i, j \in I(u)$ (by symmetry only tuples where $i \leq j$ are needed).

Even though most four-tuples are unlikely to lead to re-identification, a truly unique item-rating four-tuple could allow linking all of the items of the contributing user. To minimize this possibility we pursue three complementary approaches. First, only a random set of four-tuples is sent by each user, capped in cardinality. Second, users replace a fixed fraction (10% in our experiments) of the movie identifiers in their reports with a randomly sampled one (this alone affords 2.2-differential privacy for the set of rated movies). Third, each four-tuple $(i, r_{ui}, j, r_{uj})$ is tagged with two crowd IDs, one for $(i, r_{ui})$ and one for $(j, r_{uj})$, adding a layer of nested encryption and a second shuffler to the pipeline. This way, each item-rating combination that reaches the analysis server appears more than a threshold number of times.

| # Movies | # Users | # Reports | Score (RMSE) | |
|---|---|---|---|---|
| | | | no privacy | PROCHLO |
| 200 | 90K | 1.77M | 0.9579 | 0.9595† |
| 2K | 353K | 335M | 0.9414 | 0.9420 |
| 18K | 480K | 22.6B | 0.9222 | 0.9242 |

Table 5: Utility of the Flix evaluation; lower numbers are better. (†To account for sparsity, the threshold was set to 5.)

We perform our experiments on a dataset whose characteristics precisely match that of the Netflix Prize dataset: the number of users is 480K, the number of movies is 18K, the ratings are integers between 1 and 5. Utility is measured as the root mean square error (RMSE) and reported relative to the same benchmark used as part of the competition. Two smaller datasets (200 and 2,000 movies) are selected randomly from the main set. As seen in Table 5, the RMSE with and without PROCHLO privacy is similar across datasets.

## 6. Conclusions

Although a long-standing issue, the privacy of users' software monitoring data has recently become a pressing concern. Fortunately, those concerns can be addressed in a manner that simultaneously permits high-utility analysis, is compatible with standard software engineering practice, and provides users with strong privacy guarantees.

This paper has described how to address those privacy concerns in the context of the ESA architecture, and its PROCHLO implementation. To offer good means of balancing privacy and utility, and to minimize trust, PROCHLO introduces both new cryptographic primitives and a new algorithm for oblivious shuffling, and also relies on the advanced technologies of trusted computing and differential privacy. Even so, PROCHLO remains a relatively simple, easy-to-understand system, and a straightforward realization of the ESA architecture. However, as a framework for balancing privacy and utility, ESA is flexible enough to permit many implementations, and the use of the most appropriate techniques for different data-collection and analysis scenarios.

## Acknowledgments

This paper is dedicated to the memory of Andrea Bittau, our colleague who wrote much of PROCHLO. We thank Kunal Talwar for his help analyzing Stash Shuffle's security properties. We thank the anonymous reviewers for their detailed feedback, and Martín Abadi, Johannes Gehrke, Lea Kissner, Noé Lutz, and Nicolas Papernot for their valuable advice on earlier drafts. Our shepherd, Nickolai Zeldovich, provided invaluable help with this final paper version.

## References


[1] ABADI, M. Trusted Computing, Trusted Third Parties, and Verified Communications. In *Security and Protection in Information Processing Systems* (2004), pp. 291–308.

[2] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. Tensorflow: A system for large-scale machine learning. In *Proc. of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016), pp. 265–283.

[3] ABADI, M., BONEH, D., MIRONOV, I., RAGHUNATHAN, A., AND SEGEV, G. Message-Locked Encryption for Lock-Dependent Messages. In *Advances in Cryptology–CRYPTO* (2013), pp. 374–391.

[4] ABADI, M., CHU, A., GOODFELLOW, I., MCMAHAN, H. B., MIRONOV, I., TALWAR, K., AND ZHANG, L. Deep Learning with Differential Privacy. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 308–318.

[5] ABADI, M., ERLINGSSON, Ú., GOODFELLOW, I., MCMAHAN, H. B., PAPERNOT, N., MIRONOV, I., TALWAR, K., AND ZHANG, L. On the Protection of Private Information in Machine Learning Systems: Two Recent Approaches. In *Proc. of IEEE 30th Computer Security Foundations Symposium (CSF)* (2017), pp. 1–6.

[6] AVENT, B., KOROLOVA, A., ZEBER, D., HOVDEN, T., AND LIVSHITS, B. BLENDER: Enabling Local Search with a Hybrid Differential Privacy Model. In *Proc. of the 26th USENIX Security Symposium* (2017), pp. 747–764.

[7] BASSILY, R., NISSIM, K., STEMMER, U., AND THAKURTA, A. Practical Locally Private Heavy Hitters. *CoRR abs/1707.04982* (2017). `http://arxiv.org/abs/1707.04982`.

[8] BATCHER, K. E. Sorting Networks and their Applications. In *AFIPS Spring Joint Computer Conference* (1968), vol. 32, pp. 307–314.

[9] BELLARE, M., KEELVEEDHI, S., AND RISTENPART, T. Message-Locked Encryption and Secure Deduplication. In *Advances in Cryptology—EUROCRYPT* (2013), pp. 296–312.

[10] BINDSCHAEDLER, V., SHOKRI, R., AND GUNTER, C. A. Plausible Deniability for Privacy-Preserving Data Synthesis. *PVLDB 10*, 5 (2017), 481–492.

[11] BONAWITZ, K., IVANOV, V., KREUTER, B., MARCEDONE, A., MCMAHAN, H. B., PATEL, S., RAMAGE, D., SEGAL, A., AND SETH, K. Practical Secure Aggregation for Privacy Preserving Machine Learning. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017).

[12] BULCK, J. V., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proc. of 26th USENIX Security Symposium* (2017), pp. 1041–1056.

[13] BUSE, R. P. L., AND ZIMMERMANN, T. Information Needs for Software Development Analytics. In *Proc. of the 34th International Conference on Software Engineering (ICSE)* (2012), pp. 987–996.

[14] CHAUDHRY, G., HAMON, E. A., AND CORMEN, T. H. Relaxing the Problem-Size Bound for Out-Of-Core Columnsort. In *Proc. of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2003), pp. 250–251.

[15] CHEN, R., REZNICHENKO, A., FRANCIS, P., AND GEHRKE, J. Towards Statistical Queries over Distributed Private User Data. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (2012), pp. 169–182.

[16] CHROMIUM PROJECTS. RAPPOR (Randomized Aggregatable Privacy Preserving Ordinal Responses). `https://www.chromium.org/developers/design-documents/rappor`, 2017.

[17] CITO, J., LEITNER, P., FRITZ, T., AND GALL, H. C. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (2015), pp. 393–403.

[18] CORRIGAN-GIBBS, H., AND BONEH, D. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NDSI)* (2017), pp. 259–282.

[19] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An Anonymous Messaging System Handling Millions of Users. In *Proc. of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 321–338.

[20] DANG, H., DINH, T. T. A., CHANG, E.-C., AND OOI, B. C. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. *Proceedings on Privacy Enhancing Technologies*, 3 (2017), 18–35.

[21] DEMETRIOU, S., MERRILL, W., YANG, W., ZHANG, A., AND GUNTER, C. A. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *Proc. of the 23nd Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[22] DENNING, D. E. R. *Cryptography and Data Security*. Addison-Wesley, Boston, MA, USA, 1982.

[23] DINH, T. T. A., SAXENA, P., CHANG, E.-C., OOI, B. C., AND ZHANG, C. M2R: Enabling Stronger Privacy in MapReduce Computation. In *Proc. of the 24th USENIX Security Symposium* (2015), pp. 447–462.

[24] DOUCEUR, J. R. The Sybil attack. In *The First International Workshop on Peer-to-Peer Systems (IPTPS)* (2002), pp. 251–260.

[25] DWORK, C. Differential Privacy: A Survey of Results. In *International Conference on Theory and Applications of Models of Computation (TAMC)* (2008), pp. 1–19.

[26] DWORK, C., MCSHERRY, F., NISSIM, K., AND SMITH, A. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proc. of the Third Conference on Theory of Cryptography (TCC)* (2006), pp. 265–284.

[27] DWORK, C., AND ROTH, A. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci. 9*, 3–4 (Aug. 2014), 211–407.

[28] ERLINGSSON, Ú., PIHUR, V., AND KOROLOVA, A. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proc. of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2014), pp. 1054–1067.

[29] EVFIMIEVSKI, A., GEHRKE, J., AND SRIKANT, R. Limiting Privacy Breaches in Privacy Preserving Data Mining. In *Proc. of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (2003), pp. 211–222.

[30] FANTI, G., PIHUR, V., AND ERLINGSSON, Ú. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *Proceedings on Privacy Enhancing Technologies*, 3 (2016), 41–61.

[31] GABOARDI, M., HONAKER, J., KING, G., NISSIM, K., ULLMAN, J., AND VADHAN, S. P. PSI (Ψ): A Private data Sharing Interface. *CoRR abs/1609.04340* (2016). `http://arxiv.org/abs/1609.04340`.

[32] GANTA, S. R., KASIVISWANATHAN, S. P., AND SMITH, A. Composition Attacks and Auxiliary Information in Data Privacy. In *Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (2008), pp. 265–273.

[33] GEHRKE, J., HAY, M., LUI, E., AND PASS, R. Crowd-Blending Privacy. In *Advances in Cryptology—CRYPTO* (2012), pp. 479–496.

[34] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)* (2009), ACM, pp. 103–116.

[35] GOOGLE. gRPC: A High Performance, Open-Source Universal RPC Framework. `http://grpc.io`, 2017.

[36] Intel® Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, Oct. 2014.

[37] Intel® Software Guard Extensions (Intel® SGX) SDK. `https://software.intel.com/en-us/sgx-sdk`, 2017.

[38] JOHNSON, N., NEAR, J. P., AND SONG, D. Towards Practical Differential Privacy for SQL Queries. *CoRR abs/1706.09479* (2017). `https://arxiv.org/abs/1706.09479`.

[39] JOVIC, M., ADAMOLI, A., AND HAUSWIRTH, M. Catch Me if You Can: Performance Bug Detection in the Wild. In *Proc. of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2011), pp. 155–170.

[40] KLONOWSKI, M., AND KUTYŁOWSKI, M. Provable Anonymity for Networks of Mixes. In *Information Hiding: 7th International Workshop* (2005), pp. 26–38.

[41] KOERNER, B. I. Inside the Cyberattack That Shocked the US Government. *Wired* (Oct. 2016). `https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/`.

[42] LAURIE, B. Certificate Transparency. *Commun. ACM 57*, 10 (Oct. 2014), 40–46. `https://github.com/google/certificate-transparency`.

[43] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proc. of the 26th USENIX Security Symposium* (2017), pp. 557–574.

[44] LEIGHTON, T. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. Comput. 34*, 4 (Apr. 1985), 344–354.

[45] LI, N., QARDAJI, W., AND SU, D. On Sampling, Anonymization, and Differential Privacy or, $k$-anonymization Meets Differential Privacy. In *Proc. of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012), pp. 32–33.

[46] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable Statistical Bug Isolation. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (2005), pp. 15–26.

[47] LIE, D., AND MANIATIS, P. Glimmers: Resolving the Privacy/Trust Quagmire. In *Proc. of the 16th Workshop on Hot Topics in Operating Systems (HotOS)* (2017), pp. 94–99.

[48] MACHANAVAJJHALA, A., KIFER, D., GEHRKE, J., AND VENKITASUBRAMANIAM, M. $l$-diversity: Privacy Beyond $k$-anonymity. *ACM Trans. Knowl. Discov. Data 1*, 1 (Mar. 2007).

[49] MANIATIS, P., AND BAKER, M. Secure History Preservation Through Timeline Entanglement. In *Proc. of the 11th USENIX Security Symposium* (2002), pp. 297–312.

[50] MANIATIS, P., MIRONOV, I., AND TALWAR, K. Oblivious Stash Shuffle. *CoRR abs/1709.07553* (2017). `https://arxiv.org/abs/1709.07553`.

[51] MCSHERRY, F., AND MAHAJAN, R. Differentially-Private Network Trace Analysis. *SIGCOMM Comput. Commun. Rev. 40*, 4 (Aug. 2010), 123–134.

[52] MCSHERRY, F. D. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *Proc. of the 2009 ACM SIGMOD International Conference on Management of Data* (2009), pp. 19–30.

[53] MOHAN, P., THAKURTA, A., SHI, E., SONG, D., AND CULLER, D. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 349–360.

[54] NARAYANAN, A., AND SHMATIKOV, V. Robust De-anonymization of Large Sparse Datasets. In *Proc. of the 2008 IEEE Symposium on Security and Privacy* (2008), pp. 111–125.

[55] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Digital Signature Standard (DSS). FIPS PUB 186-2, Jan. 2000.

[56] NEWMAN, L. H. How to Protect Yourself from that Massive Equifax Breach. *Wired* (Sept. 2017). `https://www.wired.com/story/how-to-protect-yourself-from-that-massive-equifax-breach/`.

[57] OHRIMENKO, O., COSTA, M., FOURNET, C., GKANTSIDIS, C., KOHLWEISS, M., AND SHARMA, D. Observing and Preventing Leakage in MapReduce. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2015), pp. 1570–1581.

[58] OHRIMENKO, O., GOODRICH, M. T., TAMASSIA, R., AND UPFAL, E. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Part II* (2014), pp. 556–567.

[59] OLINER, A. J., IYER, A. P., STOICA, I., LAGERSPETZ, E., AND TARKOMA, S. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proc. of the 11th Conference on Embedded Networked Sensor Systems (SenSys)* (2013), ACM, pp. 10:1–10:14.

[60] PAPADOPOULOS, E. P., DIAMANTARIS, M., PAPADOPOULOS, P., PETSAS, T., IOANNIDIS, S., AND MARKATOS, E. P. The Long-Standing Privacy Debate: Mobile Websites vs Mobile Apps. In *Proc. of the 26th International Conference on World Wide Web (WWW)* (2017), pp. 153–162.

[61] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2012), pp. 107–120.

[62] REIS, C., BARTH, A., AND PIZANO, C. Browser Security: Lessons from Google Chrome. *Commun. ACM 52*, 8 (Aug. 2009), 45–49.

[63] ROY, I., SETTY, S. T. V., KILZER, A., SHMATIKOV, V., AND WITCHEL, E. Airavat: Security and Privacy for MapReduce. In *Proc. of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2010), pp. 297–312.

[64] SALTZER, J. H., AND SCHROEDER, M. D. The Protection of Information in Computer Systems. *Proc. of the IEEE 63*, 9 (1975), 1278–1308.

[65] SAMARATI, P. Protecting Respondents' Identities in Microdata Release. *IEEE Trans. on Knowl. and Data Eng. 13*, 6 (Nov. 2001), 1010–1027.

[66] SAMARATI, P., AND SWEENEY, L. Generalizing Data to Provide Anonymity when Disclosing Information (Abstract). In *Proc. of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (1998), p. 188.

[67] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proc. of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 38–54.

[68] SHAMIR, A. How to Share a Secret. *Comm. of the ACM 22*, 11 (1979), 612–613.

[69] SPENSKY, C., STEWART, J., YERUKHIMOVICH, A., SHAY, R., TRACHTENBERG, A., HOUSLEY, R., AND CUNNINGHAM, R. K. SoK: Privacy on Mobile Devices—It's Complicated. *Proc. on Privacy Enhancing Technologies*, 3 (2016), 96–116.

[70] TANG, J., KOROLOVA, A., BAI, X., WANG, X., AND WANG, X. Privacy Loss in Apple's Implementation of Differential Privacy on macOS 10.12. *arXiv:1709.02753* (2017). `https://arxiv.org/abs/1709.02753`.

[71] VALLINA-RODRIGUEZ, N., SUNDARESAN, S., RAZAGHPANAH, A., NITHYANAND, R., ALLMAN, M., KREIBICH, C., AND GILL, P. Tracking the Trackers: Towards Understanding the Mobile Advertising and Tracking Ecosystem. *CoRR abs/1609.07190* (2016). `http://arxiv.org/abs/1609.07190`.

[72] VAN DEN HOOFF, J., LAZAR, D., ZAHARIA, M., AND ZELDOVICH, N. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proc. of the 25th Symposium on Operating Systems Principles (SOSP)* (2015), pp. 137–152.

[73] VIEGA, J., CHANDRA, P., AND MESSIER, M. *Network Security with OpenSSL*, 1st ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[74] WANG, T., BLOCKI, J., LI, N., AND JHA, S. Locally Differentially Private Protocols for Frequency Estimation. In *Proc. of the 26th USENIX Security Symposium* (2017), pp. 729–745.

[75] WARNER, S. L. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *J. of the American Statistical Association 60*, 309 (1965), 63–69.

[76] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 640–656.

[77] ZHANG, L., BILD, D. R., DICK, R. P., MAO, Z. M., AND DINDA, P. Panappticon: Event-Based Tracing to Measure Mobile Application and Platform Performance. In *Proc. of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2013), pp. 33:1–33:10.

[78] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017), pp. 283–298.