# DISTRIBUTED QUERY EXECUTION WITH STRONG PRIVACY GUARANTEES

Antonis Papadimitriou

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2017

---

Andreas Haeberlen, Assistant Professor of Computer and Information Science
Supervisor of Dissertation

---

Lyle Ungar, Professor of Computer and Information Science
Graduate Group Chairperson

**Dissertation Committee**:
Jonathan Smith, Professor of Computer and Information Science
Boon Thau Loo, Associate Professor of Computer and Information Science
Aaron Roth, Associate Professor of Computer and Information Science
Ranjita Bhagwan, Researcher, Microsoft Research India

**DISTRIBUTED QUERY EXECUTION WITH STRONG PRIVACY GUARANTEES**

COPYRIGHT

2017

Antonis Papadimitriou

## Acknowledgments

First of all, I would like to thank my advisor Andreas Haeberlen for the time and effort he put to teach me and mentor me throughout these five years. Moreover, I would like to thank the members of my thesis committee for their insightful comments, useful suggestions, and their help in improving this document.

I would also like to thank my collaborators in the three projects that comprise this dissertation. Chapter 2 describes work done during an internship at MSR India in collaboration with Ranjita Bhagwan, Nishanth Chandran, Ram Ramjee, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Chapter 3 was joint work with Arjun Narayan, a senior PhD student at the time, who identified two of the most prominent systemic risk algorithms from the economics literature, and performed the sensitivity and utility analyses in the application of differential privacy in that domain. The work in Chapter 4 was done jointly with Min Xu and Ariel Feldman from the University of Chicago. In particular, Min implemented the private memory abstraction, he carried out the experiments, and we jointly developed the oblivious operators and the experimental design.

Finally, I would like to thank the people that provided the much needed emotional support to my academic endeavors. My mother, who taught me the value of education from a young age; my father, who instilled the necessary resolve in me to pursue demanding goals; and my beloved wife, whose patience, character, and love for life, made our time in Philadelphia worthwhile and memorable.

ABSTRACT

DISTRIBUTED QUERY EXECUTION
WITH STRONG PRIVACY GUARANTEES

Antonis Papadimitriou

Andreas Haeberlen

As the Internet evolves, we find more applications that involve data originating from multiple sources, and spanning machines located all over the world. Such wide distribution of sensitive data increases the risk of information leakage, and may sometimes inhibit useful applications. For instance, even though banks could share data to detect systemic threats in the US financial network, they hesitate to do so because it can leak business secrets to their competitors. Encryption is an effective way to preserve data confidentiality, but eliminates all processing capabilities. Some approaches enable processing on encrypted data, but they usually have security weaknesses, such as data leakage through side-channels, or require expensive cryptographic computations.

In this thesis, we present techniques that address the above limitations. First, we present an efficient symmetric homomorphic encryption scheme, which can aggregate encrypted data at an unprecedented scale. Second, we present a way to efficiently perform secure computations on distributed graphs. To accomplish this, we express large computations as a series of small, parallelizable vertex programs, whose state is safely transferred between vertices using a new cryptographic protocol. Finally, we propose using differential privacy to strengthen the security of trusted processors: noise is added to the side-channels, so that no adversary can extract useful information about individual users. Our experimental results suggest that the presented techniques achieve order-of-magnitude performance improvements over

previous approaches, in scenarios such as the business intelligence application of a large corporation and the detection of systemic threats in the US financial network.

# Contents

# List of Tables

# List of Figures

# 1

# Introduction

Privacy concerns are often a stumbling block to using sensitive data, in domains like healthcare [93, 52], finance [66], and social science [104, 64]. Analysis of this type of data can provide important insights, which could be used to provide high-quality user experiences, or to assist social life in diverse ways. However, this data conveys information regarding people's private lives, or corporations' business secrets. Therefore, it is reasonable for involved parties to hesitate to contribute their data for analysis, at least without getting strong guarantees that their data will not be exposed.

Consider, for instance, the detection of systemic threats in a financial network. Economists have come up with several algorithms to measure the risk of such threats [61, 63], and these algorithms could facilitate the early detection and prevention of events similar to the 2008 financial crisis. Unfortunately, these algorithms are difficult to use in practice because they require access to sensitive data from all financial institutions. Since this data reflects business secrets that give individual banks their competitive advantage, no bank is willing to share this data, and such an

application has yet to be realized. Similar settings emerge in other domains too; for example, users might be hesitant to contribute their medical history, even though this could allow medical researchers, government agencies, or even the general public, to understand nationwide health trends [93, 52].

These privacy concerns are often aggravated by the way data is distributed among many different parties. For example, in the applications described above, data may be stored at multiple locations (e.g., banks, or user devices), queries may originate from multiple sources (e.g., researchers, agencies), and computations may be executed using one or more third-party cloud services. The security of such a distributed system is as good as its weakest link, so security guarantees are harder to attain: even if one of the servers involved gets compromised by an attacker, sensitive data of all parties can be exposed.

Therefore, the key question this dissertation tries to address is how to build distributed data analytics systems, that can perform queries without leaking sensitive data, even when the involved parties are untrusted. These systems should have two important properties: (1) they should provide strong privacy guarantees, and (2) they should have practical performance by today's standards.

## 1.1 Existing approaches

One of the most widely-used ways of protecting the confidentiality of data is encryption. However, encryption severely limits the ways one can process ciphertext data. In this section, we examine four existing approaches for running computations on encrypted data.

The first approach is to use *fully homomorphic encryption (FHE)* [71]. FHE can be used to perform any desired function directly on encrypted data. Despite being very powerful, this scheme is not appropriate for our setting yet, because computations on FHE ciphertexts are many orders of magnitude slower than plaintext computations [73, 74].

*Encrypted databases* [138, 157, 118] is the second approach one could consider using to run computations on encrypted data. Encrypted databases combine several *partially homomorphic encryption* schemes to support a subset of SQL queries directly on encrypted data. Unlike FHE, partially homomorphic schemes cannot support general computations, but they are efficient enough to be practical in certain settings. However, some of the encryption schemes used in encrypted databases have some disadvantages – some of them provide weaker security guarantees, and some others incur costs that render them inefficient for large databases.

The third way to run computations on untrusted machines is to use *secret sharing* [148] and *multi-party computation (MPC)* [77, 30]. In a secret-sharing scheme, data is split into $N$ random shares, and each of the shares is given to a different party. Secret sharing provides confidentiality when at most $N-1$ parties can collude in the system (*non-collusion assumption*) because the data requires all $N$ shares to reconstruct. More importantly, the parties holding the shares can use an MPC protocol to perform any desired computation directly on the distributed shares, without ever decrypting the original value. Unfortunately, MPC protocols prevent information leakage during the execution of the protocol; if the result of a computation is published, sensitive information could still leak. Additionally, MPC quickly becomes impractical for large computations or number of parties.

The final approach for processing encrypted data is *trusted hardware* [116]. In this approach, secret cryptographic keys are integrated into the circuitry of a processor during its manufacturing, and these keys are used to establish a *trusted execution environment (TEE)* on an untrusted platform. Such an environment, sometimes called a *trusted enclave*, makes sure that data is always encrypted before it leaves the CPU boundaries, and access to the plaintext data inside the CPU is explicitly protected by the hardware itself. Assuming that adversaries do not have the expertise to decapsulate the CPU package and mount a circuit-probe attack (*trusted hardware assumption*), trusted enclaves provide the guarantee that sensitive data is never visible

by untrusted software running on the same machine, no matter what its privilege level is. In spite of making a stronger assumption, trusted hardware solutions seem unable to deliver strong privacy guarantees either; recent attacks have shown that they suffer from side-channel attacks [162, 42, 106], that is, attacks that exploit data-dependent differences in the execution of a trusted enclave to infer sensitive information.

## 1.2 Seabed

In Chapter 2, we target the setting of encrypted databases, where data and queries come from a single trusted party. As explained earlier, encrypted databases use several encryption schemes to support different kinds of operations on encrypted data. In our work, we identify that the performance bottleneck of encrypted databases lies in the asymmetric cryptographic scheme that enables additions on encrypted data. To overcome the bottleneck, we describe *additive symmetric homomorphic encryption* (ASHE), a scheme which leverages symmetric cryptographic primitives to support much faster aggregation on encrypted data. Furthermore, we observe that the efficiency of some of the remaining cryptographic schemes comes at the cost of weakened security guarantees. Indeed, recent work [125] has shown how to exploit these weaker schemes: one can examine certain patterns that emerge in the encrypted database, and use them to recover sensitive information. To mitigate this problem, we devise a scheme called SPLASHE, which transforms the database in a way that enables a large class of queries using only semantically secure encryption schemes. This is enough to prevent the emergence of ciphertext patterns, and the attacks described in [125].

We implemented these techniques in Seabed, a system that supports big data analytics on encrypted datasets. Seabed introduces several optimizations to efficiently handle performance costs associated with ASHE, as well as storage costs associated with SPLASHE. We leveraged Seabed to implement an analytics application on an

encrypted dataset from Microsoft's Bing-Ads infrastructure. Our evaluation showed that encryption in Seabed incurs minimal overhead (at most 44%) compared to the plaintext version, whereas previous approaches would incur an overhead of about one order of magnitude. Moreover, in this application, Seabed completely avoids the use of weaker encryption schemes, thereby providing strong semantic security.

## 1.3 DStress

In the second part of the dissertation (Chapter 3), we consider the general setting, where data and queries originate from multiple sources. In particular, we focus on performing graph computations, where graphs are distributed across multiple parties, but the computations require access to the entire graph. In this setting, we make the non-collusion assumption, and use secret sharing to distribute the sensitive graph data, and MPC to run graph computations on the secret-shares. To make MPC scale for real-world graphs, we express complex graph computations as simple vertex programs; these small programs are very efficient in MPC, and they can be parallelized to speed up execution. To prevent the result of the graph computation from leaking sensitive information about the graph, we add noise to the result, and use *differential privacy* [60] to select the appropriate amount of noise, so as to get strong privacy guarantees.

To evaluate this approach, we built DStress, a system for secure graph processing that supports vertex programs. One challenge that comes up after breaking up a large MPC graph computation in smaller vertex programs is that state has to be transferred from one vertex to another outside the MPC protocol. This means that some communication patterns may be visible to system participants, and could be used to infer whether an edge exists in the graph. To mitigate this problem, we devised a cryptographic protocol that transfers state between vertices, without revealing the existence of edges. We evaluated the performance of DStress by running graph algorithms that measure systemic risk in financial networks. DStress can perform

these computations on the entire US financial network in just a few hours, without leaking sensitive bank data to any participant. The same computation would take years, if banks were to perform the large graph computation in one monolithic MPC run.

## 1.4 Hermetic

Finally, in Chapter 4, we consider database query processing in the multiple-data-source setting, and under the trusted hardware assumption. In joint work with Min Xu and Ariel Feldman from the University of Chicago, we developed two techniques that aim at closing the side channels of trusted processors.

The fist technique is to create an oblivious execution environment inside the trusted CPU, which eliminates side channels that may leak sensitive information to untrusted processes running in the trusted CPU. To accomplish this, we use a trusted hypervisor to realize a *private memory abstraction*, where computations can be performed with strict isolation.

Our second technique aims at defending against traffic-analysis attacks: recent research has shown that observing the traffic produced by a distributed application can leak information [129]. Unfortunately, even though such attacks could be addressed by always padding the output of programs to the maximum possible size, this approach induces worst-case performance at all times – for example, join results should always have the cartesian product's size. To avoid this overhead, we use *differentially private padding*; with this approach, one adds just enough fake records to the result, so that the side channel is noised. The advantage of this approach is that, with differential privacy, we can compute exactly how much noise we need to get strong privacy guarantees, and, in practice, this noise turns out to be much less than the worst-case maximum. Moreover, this introduces a tradeoff between performance and privacy: the more noise we add, the better privacy is protected, and the greatest the performance hit in query execution. Therefore, in Chapter 4, we also

present a query planner which finds optimal query plans in terms of both privacy and performance.

To evaluate our design, we built a system called Hermetic, which uses the above techniques. Min implemented the Hermetic hypervisor that realizes the private memory abstraction, and provided an initial implementation of the relational primitives from [13] that we based our work on. The Hermetic runtime that executes queries and obliviously adds differentially-private noise to intermediate results, and the privacy-aware planner of Hermetic were implemented by myself. We evaluated our Hermetic prototype on a relational database of taxi trips with sensitive geolocation data, and we found out that Hermetic can complete a three-way join query in about 15 minutes, whereas the same query on a system using full padding could not complete in 7 hours. A paper with our results is currently in submission [133].

## 1.5 Contributions and road map

In this dissertation proposal, we make the following contributions:

- We introduce additive symmetric homomorphic encryption (ASHE) to enable fast aggregation on encrypted data (Section 2.3.1).
- We describe splayed additive symmetric homomorphic encryption (SPLASHE) to support an important class of aggregation queries without leaking information (Section 2.3.3).
- We report on the implementation and evaluation of Seabed, a system that leverages ASHE and SPLASHE to support big data analytics on encrypted data (Sections 2.4 and 2.6).
- We present a way to express graph algorithms as vertex programs, so that we can optimize their execution in MPC (Section 3.3.1).
- We detail a novel cryptographic protocol that can be used to transfer MPC state between vertices, without leaking information about the topology of the

distributed graph to the participants of the computation (Section 3.3.5).

- We implement DStress, a system for privacy-preserving computations on sensitive graphs distributed across multiple parties (Section 3.3), and we demonstrate that breaking up the graph computations can bring great performance benefits in real-world applications (Section 3.5).
- We describe how one can use a thin hypervisor to realize a private memory abstraction on modern processors (Section 4.3).
- We use this private memory to implement relational operators whose outputs can be padded with fake records based on principles from the differential privacy literature (Section 4.4).
- We show how differentially private padding creates new challenges in query planning, and describe a query planner that can optimize query performance and privacy (Section 4.5).
- We present the design of a system called Hermetic which uses the private memory abstraction and differentially-private padding (Section 4.6).
- We experimentally verify the security properties of an implementation of Hermetic, and report on its performance characteristics (Sections 4.7 and 4.8).

The work on Seabed, i.e., the first three contributions, started as part of an internship at MSR India, and in collaboration with Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. The results were first published in [134]. The work on DStress, i.e., the next three contributions, first appeared in [135]. Finally, the work on Hermetic, i.e., the last five contributions, resulted from joint work with Ariel Feldman and Min Xu from the University of Chicago; a paper with the results is currently in submission [133].

# 2

# Queries on data from a single source

## 2.1 Introduction

Consider a retail business that has customer and sales records from various store locations across the world. The business may be interested in analyzing these records – perhaps to better understand how revenue is growing in various geographic locations, or which demographic segments of the population its customers are coming from. To answer these questions, the business might rely on a Business Intelligence (BI) system, such as PowerBI [2], Tableau [5], or Watson Analytics [6]. These systems can scale to large data sets, and their turnaround times are low enough to answer interactive queries from customers. Internally, they rely on the cloud to provide the necessary resources at relatively low cost.

However, storing sensitive business data on the cloud can raise privacy concerns, which is why many enterprises are reluctant to use cloud-based analytics solutions. These concerns could be mitigated by keeping the data in the cloud encrypted, so that a data leak (e.g., due to a hacker attack or a rogue administrator) would cause

little or no damage. Systems like CryptDB [138] and Monomi [157] can accomplish this by using a mix of different encryption schemes, including deterministic encryption schemes [28] and partially homomorphic cryptosystems; this allows certain computations to be performed directly on encrypted data. However, this approach has two important drawbacks. First, these cryptosystems have a high computational cost. This cost is low enough to allow interactive queries on medium-size data sets with perhaps tens of gigabytes, but many businesses today collect terabytes of data [31, 94, 105, 156]. Our experimental results show that, at this scale, even on a cluster with 100 cores, it would take hundreds of seconds to process relatively simple queries, which is too slow for interactive use. Second, deterministic encryption is vulnerable to frequency attacks [125], which can cause some data leakage despite the use of encryption.

This Chapter makes two contributions towards addressing these concerns. First, we observe that existing solutions typically use *asymmetric* homomorphic encryption schemes, such as Paillier [132]. This is useful in scenarios where the data is produced and analyzed by different parties: Alice can encrypt the data with the public key and upload it to the cloud, and Bob can then submit queries and decrypt the results with the private key. However, in the case of business data, the data producer and the analyst typically have a trust relationship – for instance, they may be employees of the same business. In this scenario, it is sufficient to use *symmetric* encryption, which is much faster. To exploit this, we construct a new *additively symmetric homomorphic encryption* scheme (or, briefly, ASHE), which is up to three orders of magnitude more efficient than Paillier.

Our second contribution is a defense against frequency attacks based on auxiliary information – a type of attack that has recently been demonstrated in the context of deterministic encryption [125]. For instance, suppose the data contains a column, such as gender, that can take only a few discrete values and that has been encrypted deterministically. If the attacker knows which gender occurs more frequently in the

data, she can trivially decode this column based on which ciphertext is the most common. We introduce an encryption scheme called *Splayed ASHE (SPLASHE)*, that protects against such attacks by splaying sensitive columns to multiple columns, where each new column corresponds to data for each unique element in the original column. For columns with larger cardinality, SPLASHE uses a combination of splaying and deterministic encryption padded with spurious entries to defeat frequency attacks while still limiting the storage and computational overhead.

We also present a complete system called *Seabed* that uses ASHE and SPLASHE to provide efficient analytics over large encrypted datasets. Following the design pattern in earlier systems, Seabed consists of a client-side planner and a proxy. The planner is applied once to each new data set; it transforms the plain-text schema into an encrypted schema, and it chooses suitable encryption schemes for each column, based on the kinds of queries that the user wants to perform. The proxy transparently rewrites queries for the encrypted schema, it decrypts results that arrive from the cloud, and it performs any computations that cannot be performed directly on the cloud. Seabed contains a number of optimizations that keep the storage, bandwidth, and computation costs of ASHE low, and that make it amenable to the hardware acceleration that is available on modern CPUs.

We have built a Seabed prototype based on Apache Spark [4]. We report results from an experimental evaluation that includes running both AmpLab's Big Data Benchmark [1] and a real, advertising-based analytics application on the Azure cloud. Our results show that, compared to no encryption, Seabed increases the query latency by only 8% to 45%; in contrast, state-of-the-art solutions that are based on Paillier (such as Monomi [157]) would cause an increase by one to two orders of magnitude in query latency.

To summarize, we make the following four contributions in this Chapter:

- ASHE, an additive symmetric homomorphic encryption scheme that is three orders of magnitude faster than Paillier (Section 2.3.1);

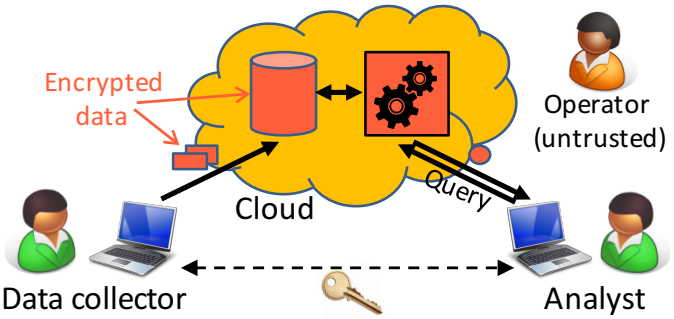


Figure 2.1: Motivating scenario.

- SPLASHE, an encryption scheme that protects against frequency-based attacks for fields that require deterministic encryption (Sections 2.3.3+2.3.4);
- Seabed, a system that supports efficient analytics over large-scale encrypted data sets (Section 2.4); and
- a prototype implementation and experimental evaluation of Seabed (Section 2.6).

## 2.2 Overview

Figure 2.1 shows the scenario we are considering in this Chapter. A *data collector* gathers a large amount of data, encrypts it, and uploads it to an untrusted cloud platform. An *analyst* can issue queries to a query processor on the cloud. The responses will be encrypted, but the analyst can decrypt them with a secret key she shares with the data collector.

The workload we wish to support consists of OLAP-style queries on big data sets. As our analysis in Section 2.5 will show, these queries mostly rely on just a few simple operations (sum, sum-of-squares, etc.), so we focus on these in our server-side design. Our goal is to answer typical BI queries on large data sets within a few seconds – that is, quickly enough for interactive analysis.

### 2.2.1 Background

One common approach to solving the above problem is to use *homomorphic encryption*. For instance, there are cryptosystems with an additive homomorphism, such as Paillier [132], which means that it is possible to "add" two ciphertexts $C(x)$ and $C(y)$ to obtain a ciphertext $C(x+y)$ that decrypts to the sum of the two encrypted values. This feature allows the cloud to perform aggregations directly on the encrypted data. There are other systems with different homomorphisms, and even *fully homomorphic* systems [71] that can be used to compute arbitrary functions on encrypted data (Section 2.7).

Homomorphic encryption schemes are typically randomized, that is, there are many different possible ciphertexts for each value. These schemes enjoy standard semantic (or CPA) security, which informally means that no adversary can learn any information about the plaintext, even given the ciphertext.

However, there are situations where it is useful to let the cloud see some property of the encrypted values (property-preserving encryption). For instance, to compute a join, the cloud *needs* to be able to match up encrypted values, which randomization would prevent. In this case, one can use *deterministic encryption* [28], where each value $v$ is mapped to exactly one ciphertext $C(v)$. However, such schemes are susceptible to *frequency attacks* [125]: if a column can only take a small number of values (say, country), and the cloud knows that some value (say, Canada) will be the most common in the data, it can look for the most common ciphertext and infer that this ciphertext must decrypt to that value. Another example of an operation achievable by a property-preserving encryption scheme is selecting rows based on a range of values (say, timestamps) in an encrypted column. Here, one can use an *order-preserving encryption (OPE)* [36], which can be used to decide whether $x < y$, given only $C(x)$ and $C(y)$. Obviously, if the cloud can perform the comparison, then so can the adversary, so in these schemes, there is a tradeoff between confidentiality, performance, and functionality.

### 2.2.2 Threat Model

In this part of the dissertation, we resolve the above tradeoff in favor of confidentiality and performance. We assume an adversary who is honest but curious (HbC), that is, the adversary will try to learn facts about the data but will not actively corrupt data or otherwise interfere with the system. We do, however, assume that the adversary will attempt to perform frequency attacks as discussed above; this is motivated by recent work [125], and it is the reason we developed SPLASHE.

We are aware that there are much stronger threat models that would prevent the adversary from learning anything at all about the data. However, current solutions for these models, such as using oblivious RAM [131, 76] and fully homomorphic encryption, tend to have an enormous runtime cost (fully homomorphic encryption [71] causes a slowdown by nine orders of magnitude [73]). Our goal is to provide a practical alternative to today's plaintext-based systems (which offer very little security), and this requires keeping the runtime overhead low.

### 2.2.3 Alternative approaches

As discussed in Section 2.2.1, one possible approach to this problem is to use homomorphic encryption. This approach is taken by systems like CryptDB [138] and Monomi [157], which use Paillier as an additive homomorphic scheme. While Paillier is *much* faster than fully homomorphic encryption, it is still expensive. For example, a single addition in Paillier on modern hardware takes about 4 $\mu$s (Section 2.4), so the latency for operations on billions of rows can easily reach several minutes.

An alternative approach is to rely on trusted hardware, such as Intel's SGX [116] or ARM's TrustZone [15]. This approach has a much lower computational overhead, but it introduces new trust assumptions that may not be suitable for all scenarios [49, 53]. It would be good to have options available that offer a low overhead without relying on trusted hardware.

### 2.2.4 Our approach

In Seabed, we solve this problem by replacing Paillier with a specially designed *additively symmetric homomorphic encryption* (ASHE) scheme. Since symmetric encryption schemes tend to be much more efficient than asymmetric schemes, this yields a big performance boost (Section 2.4). Symmetric encryption imposes a restriction that the encrypted data can only be uploaded by someone who has the secret key but this is not a constraint for the typical BI scenario. Thus, the additional protections of asymmetric cryptography are actually superfluous, and the performance gain is essentially "free".

Additionally, in order to protect against frequency attacks that occur when using deterministic or order preserving encryption, we construct a randomized encryption scheme – *SPLayed ASHE*, or SPLASHE that can still enable us to perform many queries on encrypted data that in prior work required deterministic encryption, but without leaking any information on frequency counts. Finally, for those queries that SPLASHE cannot support (e.g., joins), we support deterministic and OPE schemes that leak (a small amount of) information about the underlying plaintext values; we take this decision with the performance of the system in mind.

## 2.3 Seabed Encryption Schemes

In this section, we describe the ASHE and SPLASHE schemes in more detail. ASHE and the basic variant of SPLASHE satisfy the standard notion of semantic security (IND-CPA, that leaks no information about plaintext values) while the enhanced variant of SPLASHE provably leaks no more information than the number of dimension values that occur frequently and infrequently in the database. A formal security proof is available in Appendix A.1.

### 2.3.1 ASHE

ASHE assumes that plaintexts are from the additive group $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$. It also assumes that the entities encrypting and decrypting a ciphertext (the sender and the recipient, respectively) share a secret key $k$, as well as a pseudo-random function (PRF) $F_k : I \to \mathbb{Z}_n$ that takes an identifier from a set $I$ and returns a random number from $\mathbb{Z}_n$.

One possible choice for the PRF is $F_k := H(i||k) \bmod n$ for $i \in I$, where $H$ is a cryptographic hash function (when modeled as a random function), $||$ denotes concatenation and the size of the range of $H$ is a multiple of $n$. Another choice is AES, when used as a pseudo-random permutation.

Suppose Alice wants to send a value $m \in \mathbb{Z}_n$ to Bob. Then Alice can pick an arbitrary, unique, number $i \in I$ – which we call the *identifier* – and encrypt the message by computing:

$$\mathsf{Enc}_k(m, i) := ((m - F_k(i) + F_k(i-1)) \bmod n, \{i\})$$

In other words, the ciphertext is a tuple $(c, S)$, where $c$ is an element of the group $\mathbb{Z}_n$ and $S$ is a multiset of identifiers. Note that the ciphertext $c$ consists of the plaintext value $m$ plus some pseudo-random component, hence it appears to be random to anyone who does not know the secret key $k$.

To create the additive homomorphism, we define a special operation $\oplus$ for "adding" two ciphertexts:

$$(c_1, S_1) \oplus (c_2, S_2) := ((c_1 + c_2) \bmod n, S_1 \cup S_2)$$

That is, the group elements are added together and the multisets of identifiers are combined. To decrypt the ciphertext, Bob can simply compute

$$\mathsf{Dec}_k(c, S) := (c + \sum_{i \in S} (F_k(i) - F_k(i-1))) \bmod n$$

Thus, after the homomorphic operation,

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m_1, i_1) \oplus \mathsf{Enc}_k(m_2, i_2)) = (m_1 + m_2) \bmod n$$

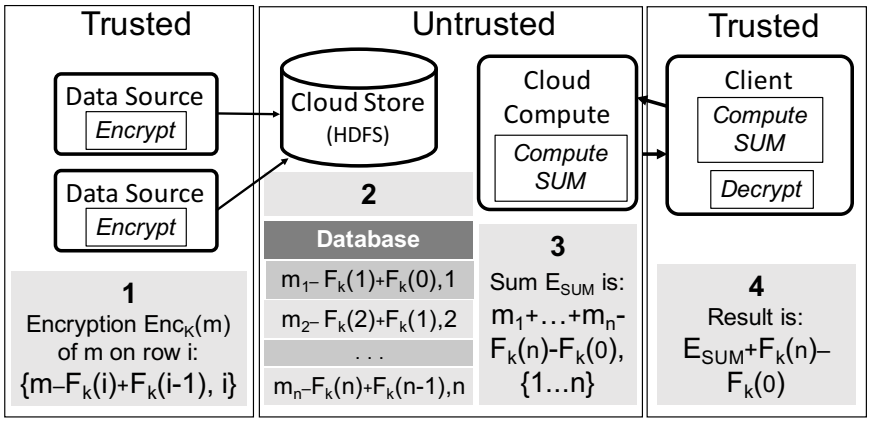



Figure 2.2: Seabed components and the ASHE scheme.

Figure 2.2 gives a high-level overview of ASHE in the context of Seabed. We show that the above scheme satisfies the standard notion of semantic (CPA) security in Appendix A.1.1.

### 2.3.2 Optimizations for ASHE

The reader may wonder why the first element of the ciphertext is computed as $(m - F_k(i) + F_k(i-1)) \bmod n$ and not simply as $(m - F_k(i)) \bmod n$. The reason is that we have optimized ASHE for computing aggregations on large data sets. Suppose, for instance, that Alice wants to give Charlie a large table of encrypted values, with the intention that Charlie will later add up a range of these values and send them to Bob. Then Alice can simply choose the identifiers to be the row of numbers $(1,2,\ldots,x)$. Later, if Bob receives an encrypted sum $(c,S)$ with $S = \{i,\ldots,i+t\}$ (i.e., the sum of rows $i$ to $i+t$), he can decrypt it simply by computing $(c + F_k(i+t) - F_k(i-1)) \bmod n$, since the other $F_k$ values will cancel out. Thus, it is possible to decrypt the sum of a range of values by evaluating the PRF only twice, regardless of the size of the range.

Other optimizations including managing ciphertext growth and use of AES encryption support in hardware for efficient PRF computation are discussed in Section 2.4.

Plaintext Schema

| country | salary |
|---|---|
| Male | 1000 |
| Female | 2000 |
| Female | 200 |

Encrypted Schema

| sender | salary |
|---|---|
| DET(Male) | ASHE(1000) |
| DET(Female) | ASHE(2000) |
| DET(Female) | ASHE(200) |

Schema with Basic SPLASHE

| genderMale | genderFemale | salaryMale | salaryFemale |
|---|---|---|---|
| ASHE(1) | ASHE(0) | ASHE(1000) | ASHE(0) |
| ASHE(0) | ASHE(1) | ASHE(0) | ASHE(2000) |
| ASHE(0) | ASHE(1) | ASHE(0) | ASHE(200) |

Figure 2.3: SPLASHE instead of deterministic encryption.

### 2.3.3 Basic SPLASHE

SPLASHE is motivated by frequency attacks on deterministic encryption [125]. Recall that, unlike ASHE, in deterministic encryption, there is only one possible ciphertext value for each plaintext value. This enables the server to perform equality checks but also reveals frequency of items. The attacker combines the frequency of ciphertexts with auxiliary information to decode them.

We begin by describing a basic version of our approach. Consider a column $C_1$ that can take one of $d$ discrete values and let the value of $C_1$ in row $t$ be $C_1[t]$. If we anticipate counting queries of the form `SELECT COUNT(`$C_1$`) WHERE` $C_1$`=x`, we can replace the column $C_1$ with a family of columns $C_{1,1},\ldots,C_{1,d}$. When the value of $C_1[t]$ is $v$, we set $C_{1,v}[t] = 1$ and set $C_{1,w}[t] = 0$ for $w \neq v$. If the resulting columns are encrypted using ASHE, the ciphertexts will look random to the adversary, but it is nevertheless possible to compute the count: we can simply rewrite the above query to `SELECT SUM(`$C_{1,x}$`)` and then compute the answer using homomorphic addition.

A similar approach is possible for aggregations. Consider a pair of columns $C_1$ and $C_2$, where $C_1$ again takes one of $d$ discrete values and $C_2$ contains numbers that we might later wish to sum up using a predicate on $C_1$ (and possibly other conditions). In other words, we anticipate queries of the form `SELECT SUM(`$C_2$`) WHERE` $C_1$`=x`. In this case, we can split $C_2$ into $d$ columns $C_{2,1},\ldots,C_{2,d}$. When $C_1[t] = v$, we set

$C_{2,v}[t] := C_2[t]$ and set $C_{2,w}[t] := 0$ for $w \neq v$. $C_1$ and $C_2$ can then be omitted. Thus, the above query can be rewritten into `SELECT SUM(`$C_{2,x}$`)`, which can be answered using homomorphic addition. An example of SPLASHE is shown in Figure 2.3 for $C_1$ as `Gender` and $C_2$ as `Salary`.

### 2.3.4 Enhanced SPLASHE

Basic SPLASHE increases a column's storage consumption by a factor of $d$, which is expensive if $d$ is large. Next, we describe an enhancement that addresses this.

Consider again a pair of columns $C_1$ (say, `country`) and $C_2$ (say, `salary`), where $C_1$ takes one of $d$ discrete values and $C_2$ contains numbers that we might later wish to sum up using a predicate on $C_1$. Suppose $k$ of the $d$ values are common (e.g., a Canadian company with offices worldwide but with most employees located in `USA` or `Canada`; $k = 2$, $d = 196$). Then we can replace $C_2$ by $k+1$ columns – one for each of the common values (`salaryUSA` and `salaryCanada`) and a single column for the uncommon values (`salaryOther`). Figure 2.4 shows an example. As before, for each row, we place the ASHE encrypted value of salary from $C_2$ in the appropriate salary column, while we fill the other $k$ salary columns with ASHE-encrypted zeros. We then encrypt $C_1$ *deterministically* for each of the uncommon countries to enable equality checks against encrypted values.

At this point it is possible to compute aggregations on $C_2$ for all values $v$ of $C_1$: if the value $v$ is common (`USA` or `Canada`), we can compute a sum over the special column for $v$; otherwise we can select the rows where `country` in $C_1$ equals the deterministically encrypted value of $v$ and compute the sum over `salaryOther`.

However, $C_1$ now is susceptible to frequency attacks. To prevent this, in $C_1$, we ensure that *all ciphertexts occur at the same frequency*. How is this possible? Note that the cells corresponding to common countries in $C_1$ were so far unused. We can reuse these cells to normalize the frequency count of the uncommon countries. For these reused cells, since the corresponding values in the `salaryOther` column

Plaintext Schema

| country | salary |
|---|---|
| USA | 100000 |
| USA | 100000 |
| Canada | 200000 |
| USA | 300000 |
| Canada | 500000 |
| Canada | 800000 |
| India | 100000 |
| India | 100000 |
| Chile | 200000 |
| Iraq | 300000 |
| China | 500000 |
| Japan | 800000 |
| Israel | 130000 |
| U.K. | 210000 |

Schema with Enhanced SPLASHE

| country | salaryUSA | salaryCanada | salaryOthers |
|---|---|---|---|
| DET(Chile) | ASHE(100000) | ASHE(0) | ASHE(0) |
| DET(Iraq) | ASHE(100000) | ASHE(0) | ASHE(0) |
| DET(China) | ASHE(0) | ASHE(200000) | ASHE(0) |
| DET(Japan) | ASHE(300000) | ASHE(0) | ASHE(0) |
| DET(Israel) | ASHE(0) | ASHE(500000) | ASHE(0) |
| DET(U.K.) | ASHE(0) | ASHE(800000) | ASHE(0) |
| DET(India) | ASHE(0) | ASHE(0) | ASHE(100000) |
| DET(India) | ASHE(0) | ASHE(0) | ASHE(100000) |
| DET(Chile) | ASHE(0) | ASHE(0) | ASHE(200000) |
| DET(Iraq) | ASHE(0) | ASHE(0) | ASHE(300000) |
| DET(China) | ASHE(0) | ASHE(0) | ASHE(500000) |
| DET(Japan) | ASHE(0) | ASHE(0) | ASHE(800000) |
| DET(Israel) | ASHE(0) | ASHE(0) | ASHE(130000) |
| DET(U.K) | ASHE(0) | ASHE(0) | ASHE(210000) |

Figure 2.4: Enhanced SPLASHE example.

are set to ASHE encrypted values of zero, this approach preserves correctness while preventing frequency attacks.

When is this approach possible? Let $n_1 \geq n_2 \ldots \geq n_d$ be the number of occurrences of each of the $d$ values. Then the number of splayed columns should be chosen to be the minimum $k$ such that $\sum_{i=1}^{k} n_i \geq \sum_{i=k+1}^{d} (n_{k+1} - n_i)$ : this is because $\sum_{i=1}^{k} n_i$ are enough unused cells in column $C_1$ that can be used to make the number of occurrences of all non-splayed values at least $n_{k+1}$. Such a $k$ will always exist; the more heavily skewed the distribution of values is, the smaller the $k$ will be, and the more storage will be saved. This approach can be followed even if the exact number of occurrences is unknown; we do, however, need to know the distribution of the values.

Figure 2.4 shows an enhanced SPLASHE example with $k = 2$ and $d = 9$. Notice how the first six rows of the deterministically encrypted column have been reused to equalize the frequency of all elements in that column while still ensuring the correctness of aggregation queries on any of the country predicates.

The reader can find a more detailed description of enhanced SPLASHE's secu-

rity properties in Appendix A.1.2. Briefly, enhanced SPLASHE satisfies simulation-based security; the adversary learns only the number of rows in the database, and the number of infrequently and frequently occurring values.

### 2.3.5 Limitations

**ASHE:** Homomorphic encryption schemes have traditionally been defined with a *compactness* requirement, which says that the ciphertext should not grow with the number of operations that are performed on it. This is done to rule out trivial schemes: for instance, one could otherwise implement an additive "homomorphism" by simply concatenating the ciphertexts $Enc(m_1)$ and $Enc(m_2)$ and then have the client do the actual addition during decryption. ASHE does not strictly satisfy compactness, but the evaluator (the cloud) still does perform the bulk of the computation on ciphertexts; also, the techniques in Section 2.4 ensure that the length of ASHE's ciphertexts does not grow too much.

In terms of performance, growing ciphertexts can create memory stress at the workers. In the case of a system without encryption, the worker nodes only need enough memory to hold the dataset. When using ASHE, the workers need to have some extra memory to construct the ID lists. This should not be a big problem in practice: as we will show in Section 2.6, the overhead is small enough for real-world big data applications that involve billions of rows. Nevertheless, this extra memory requirement can become a problem if workers have very limited memory, or if the dataset is very large (e.g., if it has trillions of records).

**SPLASHE:** SPLASHE has three main drawbacks: (1) its requirement for a-priori knowledge of query workload or data distribution, (2) its difficulty in handling data with rapidly changing distribution, and (3) its storage overhead.

First, SPLASHE requires knowing what the expected query workload is. This is because we need to confirm that the splayed column will not participate in joins or inequality predicates – for such cases we need to fall back to deterministic encryption

(DET). In addition, to get the storage reduction of enhanced SPLASHE, we need to know the distribution of values that a column can take. If this information is not available, only basic SPLASHE can be used.

Second, enhanced SPLASHE is most appropriate for columns whose distribution does not change dramatically. For columns whose distribution fluctuates significantly, data insertions will start skewing the distribution of the DET column ($C_1$ in our example) away from the uniform distribution SPLASHE constructs. This happens because a significant change in distribution will require reusing more cells than those available in the rows that were previously common. However, even in such an extreme case, SPLASHE is still better than using plain DET; DET reveals the exact distribution of values, whereas SPLASHE reveals a noised version of it.

Finally, both basic and enhanced SPLASHE increase storage needs. Section 2.6.6 shows that a real-world ad analytics database can be supported with enhanced SPLASHE at a storage overhead of about 10x.

## 2.4 Design

We now provide a functional overview of Seabed, and then describe each system component in more detail. For simplicity, we describe the design using the example of only one data source and one client. In practice, multiple data sources and users can share the same system as long as they share trust.

### 2.4.1 Roadmap

Figure 2.5 shows the major components of Seabed. A user interacts with the Seabed client proxy that runs in a trusted environment. The proxy in turn interacts with the untrusted Seabed server. As with previous systems, Seabed is designed to hide all cryptographic operations from users, so they interact with the system in the same way as they would with a standard Spark system. The user can issue three kinds of requests:

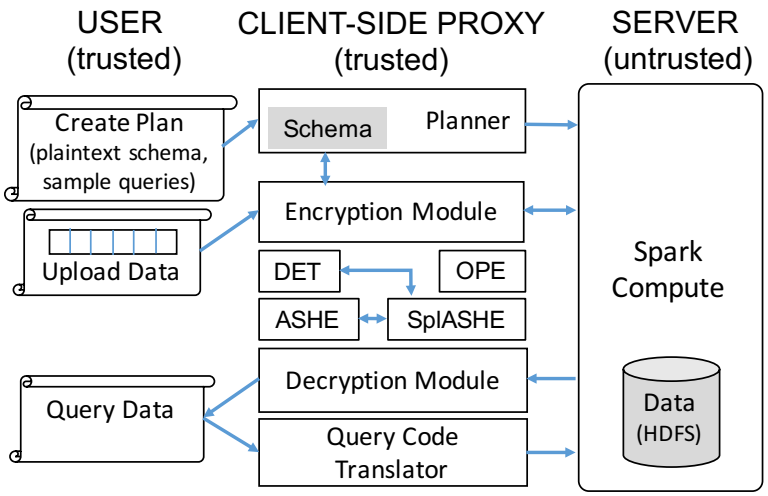



Figure 2.5: Seabed system design

**Create Plan:** First, the user supplies a plaintext schema and a sample query set to the Seabed planner. The planner uses these and the procedure specified in Section 2.4.2 to determine the encryption schemes for the columns.

**Upload Data:** Next, the user sends plaintext data to the Seabed encryption module described in Section 2.4.3. The data is encrypted with the required encryption scheme and records are appended to the table stored in the Cloud. This is a continuing process; database insertions are handled in the same way.

**Query Data:** During analysis, the user sends a query script to the Seabed query translator, which modifies queries to run on encrypted data before sending them to the server (Section 2.4.5). The server runs the queries and responds to the proxy's decryption module (Section 2.4.6). After decryption and further processing (if any), the results are sent back to the user.

### 2.4.2 Data Planner

The data planner determines how to encrypt each column in the schema, given a list of sensitive columns by the user. The user also supplies a sample query set, which is used by the planner to decide on the encryption algorithms. In addition, to use enhanced SPLASHE, the user provides the number of distinct values each column

can take and the frequency distribution of these values.

By parsing the sample query set, the planner first classifies each sensitive column as a *dimension*, a *measure*, or both. A measure is a column (e.g., `Salary`) over which a query computes aggregate functions, such as sum, average and variance. A dimension is a column (e.g., `Country`) that is used to filter rows based on a specified predicate before computing aggregates. After the classification, the planner uses the following strategies to determine which encryption schemes to use.

**ASHE:** If a sensitive measure is aggregated using linear functions, such as sum and average, we encrypt it using ASHE. If a sensitive measure is aggregated using quadratic functions (e.g., variance), we compute the square of the values on the client side and add it to the database as a separate column, so it can be used in computations on the server side. Whenever we use ASHE on a column, we give a unique ID to each row, which is used in the encryption as discussed in Section 2.3.1; to enable compression, we assign consecutive row IDs. We choose a different secret key $k$ for each new column we encrypt.

**SPLASHE:** If a sensitive dimension is used in filters, and if no query uses joins on this dimension, then the dimension is a candidate for SPLASHE. However, given the storage costs, we determine whether to use SPLASHE for the dimension as follows. First, we determine the measure columns that are used in conjunction with this dimension in the queries: only these measure columns need to be SPLASHE-encrypted. Based on this subset of measure columns, the planner uses the algorithm described in 2.3.4 to compute the storage overhead. Then, if a user specifies a maximum storage overhead, the planner prioritizes the dimensions that use SPLASHE based on their cardinality (lowest cardinal dimension first, in order to maximize protection against frequency attacks). We show how this approach works with a real dataset in Section 2.6.6.

**DET or OPE:** If a sensitive dimension cannot use SPLASHE – say, because it is used as part of a join – we warn the user and then use deterministic encryption (DET). If

| Operation | Time (nanoseconds) |
|---|---|
| AES counter mode | 47 |
| Paillier encryption | 5,100,000 |
| ASHE encryption/decryption | 12-24 |
| Plain addition | 1 |
| Paillier addition | 3800 |
| Paillier decryption | 3,400,000 |

Table 2.1: Cost of operations on a 2.2 GHz Xeon core.

the dimension requires range queries in query filters, then we use order-preserving encryption (OPE). We require an OPE scheme that works on dynamic data and hence the OPE scheme of CryptDB [138] is not suitable in our case. We use the recent scheme from [46], which is efficient (based on any PRF) and has low leakage: for any two ciphertexts, in addition to the order of the two underlying plaintexts, it reveals the first bit where the two plaintexts differ and nothing more. For more details, please see Appendix A.1.

Note that some queries (such as averages) cannot be directly executed on the server because they are not supported by Seabed's encryption schemes. In such cases, the Seabed planner borrows techniques from prior work [157] to divide the query into a part the server *can* compute (e.g., a sum and a count), and a part that the client/proxy will need to compute after decryption (e.g., the final division).

### 2.4.3 Encryption Module

The Encryption Module encrypts plaintext records into the encrypted schema. Note that ASHE encryption and decryption are quite lightweight compared to Paillier operations. As shown in Table 2.1, one AES counter operation (implemented using hardware support on a Intel Xeon 2.2GHz processor) takes 47 ns whereas one Paillier encryption takes 5.1 ms, a difference of *five orders of magnitude*. Hence, by using ASHE instead of Paillier, we reduce the encryption load on the client significantly.

We optimize ASHE encryption and decryption further by using a single AES operation to generate multiple ciphertexts. Each AES operation works on 128-bit

vectors. Numeric data types are typically much smaller: 32-bit or 64-bit integers are common. One AES operation can therefore generate two or four pseudo-random numbers for 64-bit or 32-bit data types, respectively.

Also, note that unlike conventional cryptographic techniques, ASHE encryption and decryption are inherently parallelizable because multiple AES operations can be computed simultaneously in a multi-core environment. We therefore run a multi-threaded version of the encryption and decryption algorithm, and this further reduces latency.

If the system needs a way to revoke the access privileges of individual users, the proxy can additionally implement an access control mechanism, analogous to the approach in CryptDB. Typically, revocation is difficult when symmetric encryption schemes are used: once a symmetric key is shared, the only way to invalidate it is to re-encrypt the data. However, since the proxy handles all queries, it does not need to share the secret keys with the clients, so it can revoke or limit their access without re-encryption.

### 2.4.4 Query Translator

The goal of the Query Translator is to intercept the client's unmodified queries, and rewrite them in a way appropriate for the schema of the encrypted dataset. Our design follows the principles introduced by CryptDB and Monomi: we encrypt constants with the appropriate encryption scheme, and we replace operators with the custom functions that implement ASHE aggregation, or DET/OPE checks. One technical difference to the previous systems is that these operated on relational databases, so both the source and target language of the translator was SQL. However, Seabed works on Spark, so the target language is Scala and the Spark API.

The Seabed Query Translator makes three additions to the query rewriting process to accommodate the new encryption schemes it uses; we show examples for all three in Table 2.2. First, the schema of the encrypted dataset in Seabed includes an additional ID column. This column is necessary for ASHE aggregation, so the Query

| Query type | | Query |
|---|---|---|
| ID preservation | SQL | `SELECT sum(tmp.a)`<br>`FROM (SELECT a FROM table WHERE b > 10) tmp` |
| | Spark API | `table.filter(x => x(2) > 10).`<br>`map(x =>x(1)).reduce((x,y) => x+y)` |
| | Seabed | `table.filter(x => OPE.leq(x(2),Enc`$_{OPE}$`(10)).`<br>`map(x =>(x(id), x(1))).`<br>`reduce((x,y) => ASHE(x,y))` |
| SPLASHE | SQL | `SELECT count(*) FROM table WHERE a = 10` |
| | Spark API | `table.filter(x=>x(1) == 10).count()` |
| | Seabed | `table.map(x=>(x(id),x(3))).`<br>`reduce((x,y)=>ASHE(x,y))` |
| Group-by optimization (and ID preservation) | SQL | `SELECT a, sum(b) FROM table GROUP BY a` |
| | Spark API | `table.map(x=>(x(1),x(2)).`<br>`reduceByKey((x,y)=>x+y)` |
| | Seabed | `table.map(x=>(x(1)+":"+r.nextInt%10,(x(id),x(2))).`<br>`reduceByKey((x,y)=>ASHE(x,y))` |

Table 2.2: Examples of query translation. $x(1)$ corresponds to table column $a$, $x(2)$ to $b$, $x(3)$ to splayed $a$ for value 10, and $x(id)$ to the identifier column used by ASHE.

Translator preserves it even if the client has not explicitly done so in the projection fields of the original SQL query. That way, Seabed can support aggregation on the result of sub-queries. Second, for columns that use SPLASHE, Seabed follows the rules outlined in Section 2.3 to rewrite queries. This implies that the client has to maintain a small data structure with information about the splayed fields. Finally, if the client enables our group-by optimization, which is described in Section 2.4.5, the Query Translator may also modify the group-by fields of the query. This requires that the client maintains some state about the expected number of groups in a query result.

| Technique | Example | |
|---|---|---|
| | Integer/List | Encoding |
| Range encoding | [2…14,19…23] | [2-14,19-23] |
| Diff. encoding | [2,3,4,9,23] | [2,1,1,5,14] |
| Combination | [2…14,19…23] | [2-12,5-4] |
| VB-encoding | Encoded with minimum #bytes | |

Table 2.3: ID list encoding techniques used in Seabed.

### 2.4.5 Seabed Server

Performing aggregations using ASHE requires the server to manage growing ciphertexts. This can result in need for large in-memory data structures and high bandwidth. We now describe how we optimize these overheads.

**Reducing ID list size:** To keep the size of the ID list small, we evaluated several integer list encoding techniques [107], including bitmaps [44], for good compression rates, low memory usage and high encoding speed. We eventually decided that a combination of the techniques listed in Table 2.3 were the most appropriate for Seabed. We begin with range encoding, which compresses contiguous sequences of integers by specifying the lower and upper bound. Next, we apply differential (Diff) encoding, which replaces the (potentially large) individual numbers with the (hopefully small) difference to the previous number; the result of this second step is labeled "Combination" in Table 2.3. Finally, we apply variable-byte (VB) encoding, which uses fewer bytes to represent smaller numbers.

Variable-byte (VB) and differential encoding (Diff) strike a nice balance between performance and compression and can be efficiently implemented in software. Range encoding, i.e. describing contiguous integers by specifying the bounds of their range, is not widely used in the literature because it can bloat up lists of non-contiguous integers. In Seabed, though, data is uploaded to the server with contiguous IDs, so range encoding can provide great benefits, especially for queries that select a large portion of a dataset. In Section 2.6.4, we show how combining VB, Diff, and range encoding reduces the size of the ID list and speeds up aggregation.

| Query set | Total | Purely on Server | Client Pre-processing | Client Post-processing | Two Round-trips |
|---|---|---|---|---|---|
| Ad Analytics | 168,352 | 134,298 | 0 | 34,054 | 0 |
| TPC-DS | 99 | 69 | 2 | 25 | 3 |
| MDX | 38 | 17 | 12 | 4 | 5 |

Table 2.4: Different categories of queries that Seabed supports.

**Reducing server-to-client traffic:** Every Spark job consists of one driver node and several worker nodes. The workers send their partial results to the driver which then aggregates and sends the combined result to the client. To further reduce the size of ID lists, we applied standard compression. However, there are two options here: applying compression at the worker nodes or applying compression after aggregation at the driver node. The latter can lead to higher compression rates, but we found that this caused a bottleneck at the driver. Instead, we found that applying compression at each of the worker nodes benefits from parallelization and results in lower overall latency.

**Handling group-by queries:** Group-by queries are in general challenging for ASHE, because all row IDs are included in the final result, which can grow quite large. Moreover, using range encoding seems to incur unnecessary costs for group-by queries: when the result of a group-by query contains many groups, the ID lists of each group tend to be very sparse. As we noted earlier, range encoding is wasteful for sparse ID lists, so we decided to use only VB and Diff encoding for group-by queries.

Group-by queries lead to one more complication: when the number of groups in the result is small, the traffic between mapper and reducer workers becomes a bottleneck. There are two underlying reasons for this. First, with few groups, the ID list of each group becomes denser, and not using range encoding starts to show up. Second, when the number of groups is less than the available workers, some reducers

| Dataset | Rows | Dimensions | Measures | Disk size (GB) | | |
|---|---|---|---|---|---|---|
| | | | | NoEnc | Seabed | Paillier |
| Synthetic - Large | 1.75B | - | 1 | 35.4 | 70.4 | 521.1 |
| Synthetic - Small | 250M | - | 1 | 5 | 9.8 | 74.2 |
| BDB - Rankings | 90M | 1 | 2 | 7.9 | 12 | 58.3 |
| BDB - User Visits | 775M | 8 | 2 | 194.9 | 287.5 | 673.6 |
| BDB - Query 4, Phase 2 | 194M | 2 | 1 | 35 | 38.3 | 88.3 |
| Ad Analytics | 759M | 33 | 18 | 132.3 | 142.45 | 176.3 |

Table 2.5: Characteristics and disk size of our synthetic dataset, the Big Data Benchmark (BDB) and the Ad Analytics dataset (AdA).

will remain idle in the reduce phase. This means that more data (because of denser ID lists) is shuffled between fewer workers (because of idle workers). This can create a bottleneck for very large datasets where ID lists are large.

To make use of more worker nodes in the reduce phase and to mitigate the above effect, we artificially increase the number of returned groups. We accomplish this by appending a random identifier to each value of the group-by column. For example (table 2.2), if a query returns 10 groups $\{g_1,\ldots,g_{10}\}$, and there are 100 workers available, then we can append a random identifier to the group-by column, which takes values from 0 to 9. This means that the result will contain $10*10=100$ groups $\{g_1{:}0,\ldots,g_1{:}9,\ldots,g_{10}{:}0,\ldots,g_{10}{:}9\}$, the computation will utilize all available workers in the reduce phase, and we will avoid the bandwidth bottleneck. Of course, the client has to perform the remaining aggregations to compute the sum of the actual groups (e.g., add results for groups $\{g_1{:}0,\ldots,g_1{:}9\}$ to get the result for group $g_1$). As a heuristic, we inflate the number of groups to the number of available workers when we expect fewer groups than workers.

| Dataset | Rows | Dimensions | Measures | Memory size (GB) | | |
|---|---|---|---|---|---|---|
| | | | | NoEnc | Seabed | Paillier |
| Synthetic - Large | 1.75B | - | 1 | 84.7 | 121.9 | 638.6 |
| Synthetic - Small | 250M | - | 1 | 12.1 | 17.7 | 91.4 |
| BDB - Rankings | 90M | 1 | 2 | 18.6 | 28.1 | 80.4 |
| BDB - User Visits | 775M | 8 | 2 | 581 | 832.5 | 1269.4 |
| BDB - Query 4, Phase 2 | 194M | 2 | 1 | 73.5 | 86.5 | 140 |
| Ad Analytics | 759M | 33 | 18 | 1004 | 1027.3 | 1254.4 |

Table 2.6: Characteristics and memory size of our synthetic dataset, the Big Data Benchmark (BDB) and the Ad Analytics dataset (AdA).

### 2.4.6 Decryption Module

The Decryption Module uncompresses the ID lists, uses the techniques from Section 2.4.3 to calculate the pseudo-random numbers to add to the encrypted value, and returns the result to the user. If the query has some part that cannot be computed at the server, the Decryption Module can additionally perform that part before presenting the final answer to the user. Since we have assumed that the adversary is honest but curious, the Decryption Module performs no integrity checks; thus, an active adversary could return bogus data without being detected by Seabed itself.

The decryption cost of ASHE depends on the number of aggregated elements; this is different from Paillier, which requires only one decryption for each aggregate result. However, Paillier decryption is five orders of magnitude slower than ASHE decryption (Table 2.1), and the overall client decryption costs for Seabed remain smaller than Paillier (Section 2.6).

## 2.5 Applications

An important question is whether Seabed supports a wide range of big data analytics applications. To understand this, we performed three studies. First, we systematically analyzed two common interfaces that BI applications use at the back-end: MDX (the industry standard) and Spark. Second, we evaluated a month-long query log made on a custom-designed advertising analytics OLAP platform to determine how effectively Seabed can support the functionality of these systems. Finally, we analyzed the TPC-DS query set. Detailed results of our MDX/Spark analysis can be found in Appendix A.2. Briefly, the analysis revealed that Seabed's functionality support falls into four categories:

**Support fully on the server:** Seabed's encryption techniques can fully support operations with no client support. Examples of such operations are computing the sum, average, count, and min.

**Support with client pre-processing:** Seabed can support quadratic computation necessary for more complex analytics such as anomaly detection, linear regression in one dimension, and decision trees that are supported by Watson Analytics [6] and Tableau [5]. To support this, the Seabed client has to compute squared values of the necessary columns, and encrypt them with ASHE.

**Support with client post-processing:** All applications and APIs we studied allow users to specify arbitrary functions of data. When these functions are complex, Seabed cannot perform them at the server and data has to be post-processed at the client. This is similar to how Monomi splits queries into server- and client-side components.

**Support with two client round-trips:** Some queries require the client to compute an intermediate result, re-encrypt it and send it back to the server for further processing.

Table 2.4 shows the numbers of queries that fall into these categories for the three

query sets we analyzed. We analyzed the MDX API/TPC-DS query set manually; for the ad analytics query set, we used heuristics based on the query structure. For Ad Analytics and TPC-DS, about 75-80% of the queries can be supported purely on the server. This implies that these query sets mostly use simple aggregation functions. About 20-25% need client-side support. The TPC-DS query set and MDX API have a few queries (5-15%) that require two round-trips.

## 2.6 Evaluation

In this section, we report results from our experimental evaluation of Seabed. Table 2.5 summarizes the datasets used in our experiments. We evaluate the system with microbenchmarks (Synthetic), an advertising analytics data workload and query set (AdA), and the AmpLab Big Data Benchmark (BDB).

Our evaluation has two high-level goals. First, we evaluate the *performance benefits* of Seabed over systems that use the Paillier cryptosystem. Second, we quantify the *performance and storage overhead* incurred by Seabed as compared to a system with no encryption.

### 2.6.1 Implementation and Setup

We built a prototype implementation of Seabed on the Apache/Spark platform [4] (version 1.6.0). We chose Spark because of its growing user-base and performant memory-centric approach to data processing. The server-side Seabed library was written in Scala using the Spark API. The Seabed client uses Scala combined with a C++ cryptography module for hardware accelerated AES (with Intel AES-NI instructions). We implemented Paillier in Scala using the `BigInt` class. Data tables are stored in HDFS using Google Protobuf [3] serialization. In total, our Seabed prototype consists of 3,298 lines of Scala and 2,730 lines of C++.

Our experiments were conducted on an Azure HDInsight Linux cluster. The cluster consists of tens of nodes, each equipped with a 16-core Intel Xeon E5 2.4

GHz processor and 112 GB of memory. Machines were running Ubuntu (14.04.4 LTS) and job scheduling was done through Yarn. In our experiments, we compare the following system setups:

**NoEnc:** Original Spark queries over unencrypted data,

**Paillier:** Modified Spark queries over encrypted data; measures are encrypted using Paillier, and dimensions with DET and/or OPE, and

**Seabed:** Modified Spark queries over encrypted data; measures are encrypted using ASHE, and dimensions with DET and/or OPE.

For our microbenchmarks, we generated a synthetic dataset (see Table 2.5). The NoEnc and Paillier datasets consist of one column of plaintext integers and 2048-bit ciphertexts, respectively. The ASHE dataset consists of two columns: an ID and an integer value encrypted with ASHE (IDs are contiguous). In order to model predicates that choose selected rows of a table, we use a parameter called *selectivity* that varies between 0 and 1 and use it to choose *each row* randomly with the corresponding probability. Note that this random selection model allows us to study the various system trade-offs in these schemes, e.g., the total length of ID lists, and it also enables us to understand the worst-case behavior. (At first glance, a query that selects all even or odd rows may appear to be the worst case for Seabed, since range encoding with such a non-contiguous set of IDs will double the size of the resulting ID list. However, in this case, the ID list is in fact highly compressible because the differences between consecutive IDs is always two, so stock compression techniques work very well.)

All experiments, unless otherwise mentioned, used 100 cores and 1.75 billion rows of input data. For end-to-end results, we place the client in one of the nodes in the same cluster as the server. Thus, by default, the client is connected by a high-speed, low-latency link to the server (TCP throughput of 2 Gbps). However, we also perform experiments by varying this bandwidth (using the `tc` command).

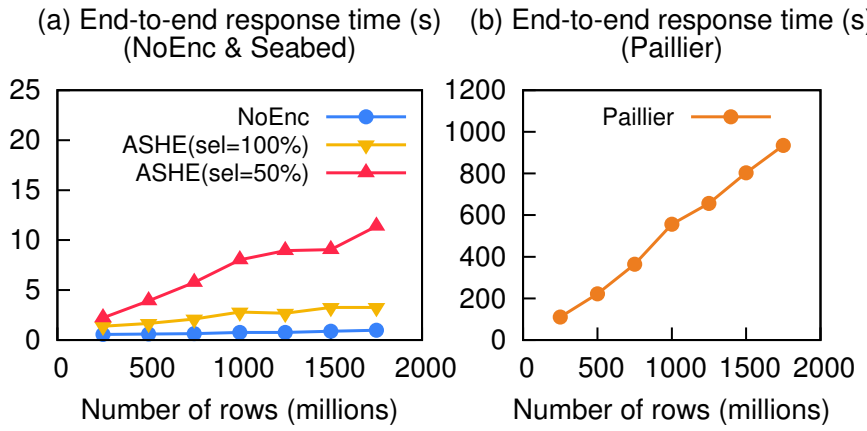



Figure 2.6: Median latency for aggregation vs data size.

### 2.6.2 Microbenchmark: End to End Latency

We first compare end-to-end latency for the three approaches with varying input sizes (250 million to 1.75 billion rows). In Figure 2.6, we show the median latency after running 10 queries for each input size. For Seabed, we show two lines: one with selectivity 100% and the other with selectivity 50%. We shall show in Section 2.6.4 that the former gives best-case latency while the latter gives worst-case latency for Seabed. For NoEnc and Paillier, we use a selectivity of 100% (their performance is linear with respect to selectivity).

Figure 2.6(a) shows the results for NoEnc and Seabed. NoEnc has a constant latency of approximately 0.6s. This is because addition is a simple operation and the overall latency is dominated by task creation costs. Seabed's aggregation is more complex, so latency for both Seabed selectivity 50% and 100% increases linearly with the dataset size. Nevertheless, the cost of aggregation in Seabed is still small even for large datasets, varying between 1.8s to 11s in the worst-case as the number of rows increase. On the contrary, Paillier results in a latency of over 1000s when aggregating 1.75 billion rows.

For Seabed selectivity 100%, about 80% of time is due to server-side compute, 20% is due to client-side decryption, and network latency is minimal. For Seabed

selectivity 50%, the server-side contributes 55% of the latency, the decryption contributes 35% and network transfer contributes the remaining 10%.

We observed occasional stragglers, i.e., tasks that took longer to complete and delayed the entire job, for all three systems. The underlying cause of these stragglers was usually garbage collection being triggered at some node in the cluster. Paillier jobs took several hundreds of seconds to complete, so the comparative effect of stragglers was small. However, NoEnc and Seabed jobs took only few seconds at the server, so whenever there was a straggler task, the delay was more pronounced.

### 2.6.3 Microbenchmark: Server Scalability

One important aspect of big data systems is how they scale with larger clusters. Since using a larger cluster can only speed up the server side, we consider *server-side latency* as we evaluate Seabed's scalability. Fixing the dataset at 1.75 billion rows, we varied the number of cores from 10 to 100. Figure 2.7 shows how Seabed, NoEnc and Paillier scaled with the number of cores. NoEnc reached its best latency, which is approximately 1s, with 20 cores. Both Seabed selectivity 100% and Seabed selectivity 50% achieved their best latency of 1.35s and 8.0s respectively with only 50 cores. Even with 100 cores, Paillier's server latency was close to 1000s, which is more than two orders of magnitude higher than Seabed's. This implies that, for large datasets, Paillier would require increasing the number of cores by orders of magnitude in order to achieve latencies that are comparable to Seabed. Seabed's overhead over NoEnc primarily comes from managing the ID lists. Next, we look into this in more detail.

### 2.6.4 Microbenchmark: Seabed Overhead

In this section we examine the server-side overheads incurred by Seabed's ASHE and the use of OPE.

**ASHE list construction:** For ASHE, the server manages ID lists using a variety of compression techniques (Section 2.4). In this experiment, we show how these com-

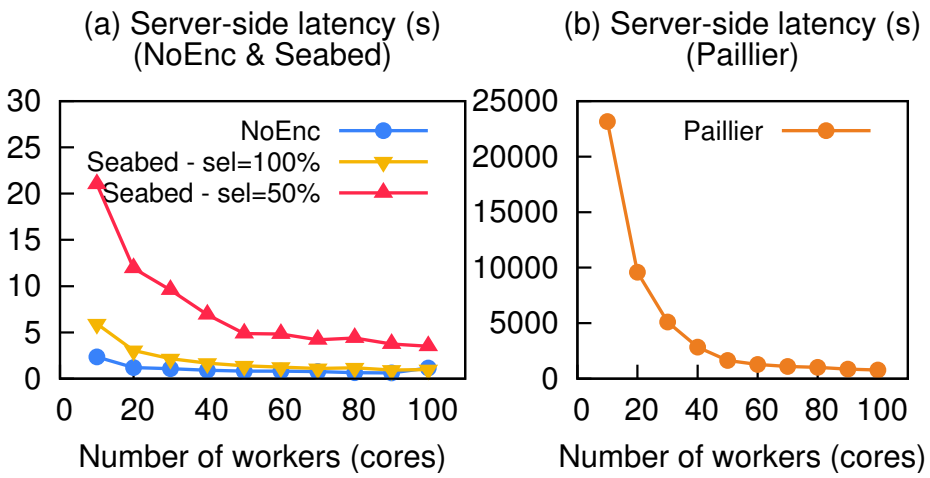



Figure 2.7: Median latency for aggregation vs cores.

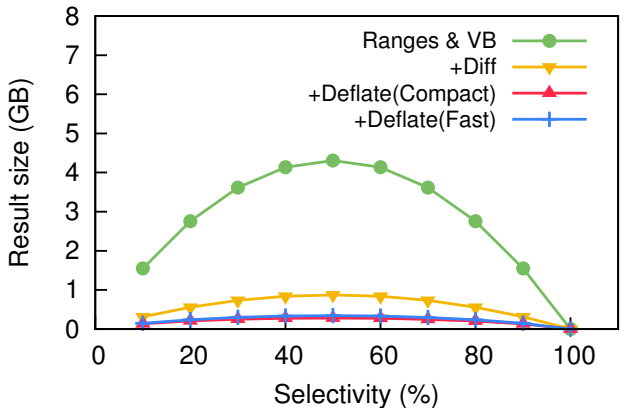



Figure 2.8: Result size vs selectivity over 1.75 billion rows.

pression techniques perform. The bitmap algorithms performed poorly, so we omit them here for brevity. We varied selectivity from 10% to 100%, and we measured the size of the ID list and the server-side response time of the query. We report the results in Figure 2.8 and 2.9.

Figure 2.8 suggests that range encoding is very effective in bounding the length of the ID list: without it, the size of ID list would keep increasing as the selectivity of a query increases, whereas with ranges the list size starts decreasing after selectivity 50%. After this, IDs start to become more dense and therefore more consecutive, leading to best-case compression at selectivity 100%. We can also see that the com-

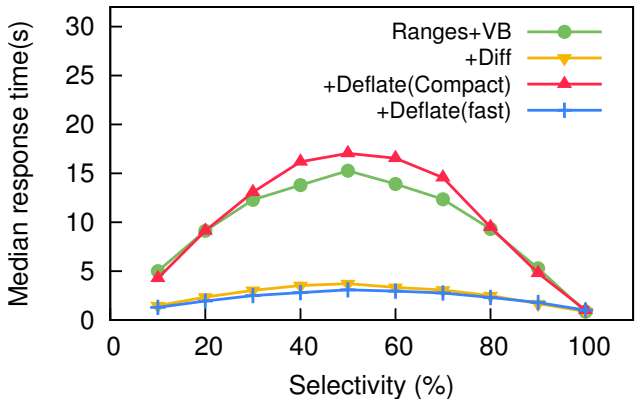



Figure 2.9: Response time vs selectivity over 1.75 billion rows.

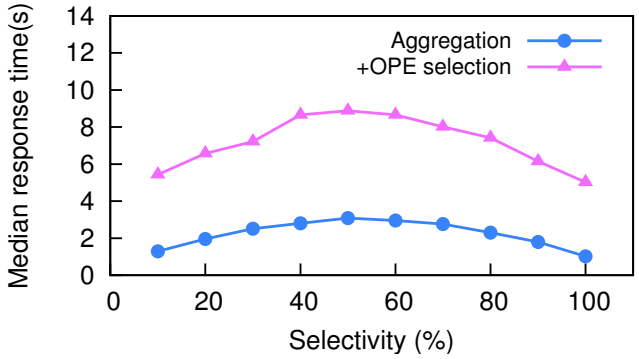



Figure 2.10: Response time vs selectivity over 1.75 billion rows, for OPE.

bination of VB and Diff-encoding is very effective in reducing the size of the ID list, and Deflate compression [54] further reduces the size of the list.

The performance hit incurred by each encoding method is depicted in Figure 2.9. To our advantage, we found that, in all cases except with Deflate optimized for high compression ratio, the better-performing algorithms also provided more compressed ID lists. Based on the above, we picked the following combination of encodings as the ID list construction algorithm in Seabed: Range-encoding, VB encoding, Diff-encoding, and Deflate compression (optimized for speed). This is what we used for all the other experiments.

**OPE:** The OPE scheme we use introduces some overhead because comparison be-

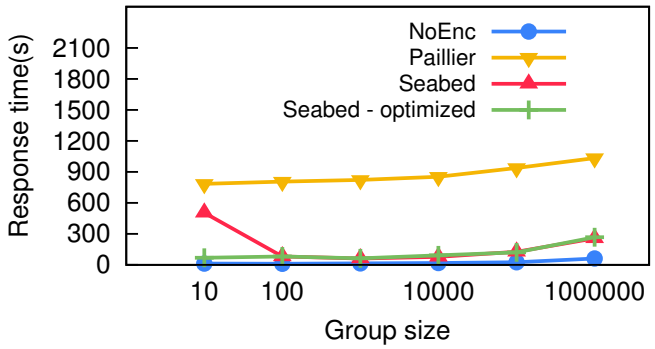



Figure 2.11: Microbenchmark results for group-by queries.

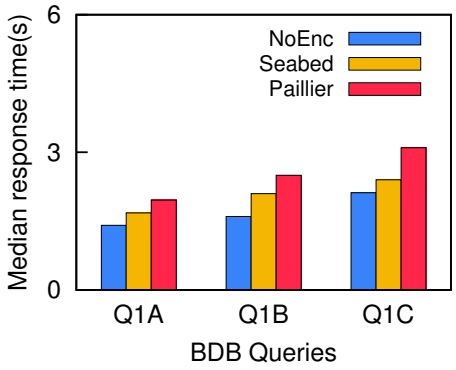



Figure 2.12: Response time for the Big Data Benchmark queries (part 1).

tween OPE ciphertexts is not as fast as comparing two plaintext integers. This is because OPE comparison involves searching for the first bit position where two 64-bit integers differ.

To measure the cost of OPE, we used the same synthetic dataset as for ASHE with 1.75 billion rows, but we added one more integer column encrypted with OPE. We repeat the selectivity experiment above, but with the query performing an OPE comparison. Figure 2.10 indicates that OPE introduces more overhead, of about a factor of 5s, compared to the ASHE ID list construction.

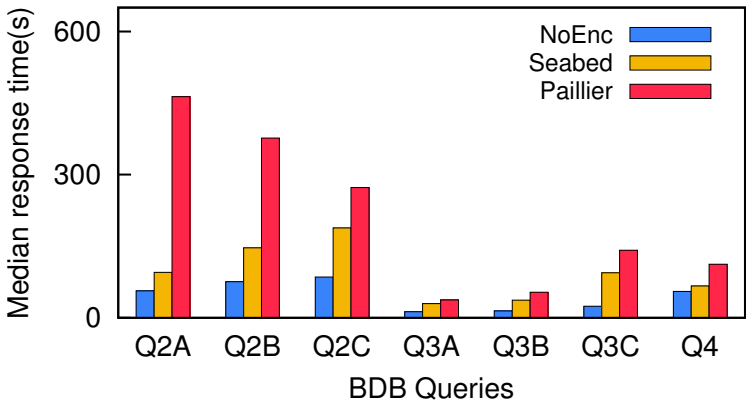



Figure 2.13: Response time for the Big Data Benchmark queries (part 2).

### 2.6.5 Microbenchmark: Group-by

So far, we have evaluated only simple aggregation queries that involved minimal network communication: each Spark worker computes a sum and a compressed ID list per partition, and the reducers concatenate the lists into the final result. While aggregation is a major component of analytical query workloads, many queries also use the group-by operation, which causes more data to be shuffled across workers. In this section, we examine how Seabed performs for queries that involve group-by.

For this experiment, we used the synthetic dataset from the previous sections, but we added one more integer column. We then aggregated the value field while doing a group-by on the new column. We varied the number of groups from 10 to 1 million; Figure 2.11 shows the results.

The Seabed line shows the performance we get when we use VB and Diff-encoding for group-by queries. A very small number of groups in the result (10 in Fig. 2.11) leads to increased latency because of the bandwidth bottleneck described in Section 2.4.5. The Seabed-optimized line shows that we can effectively deal with this inefficiency by artificially increasing the number of groups to 100 (Section 2.4.5).

Since all IDs are included in the result, Seabed group-by queries involve a significant amount of data shuffling. As a consequence, the benefits Seabed enjoys when compared to Paillier are lower. Yet, Seabed (optimized) does seem to be faster than Paillier by 5x to 10x. As the number of groups increases, Seabed's gain over Paillier drops from 10x to 5x. This is because the network shuffle time becomes a more significant part of the server response time. This indicates that Seabed will be less effective for group-by queries with a huge number of groups (hundreds of millions), something we observe in Section 2.6.6.

### 2.6.6 Ad-Analytics Workload

To assess the performance of Seabed on real-world data and queries, we evaluated it using the AmpLab Big Data Benchmark [1] and using a real-world large-scale advertising analytics application. We begin with a discussion of the latter.

For this series of experiments, we used data from an advertising analytics application deployed at an enterprise. This application is used by a team of experts for analytical tasks such as determining behavioral trends of advertisers, understanding ad revenue growth, and flagging anomalous trends in measures such as revenue and number of clicks. The data characteristics are shown in Table 2.5. We also obtained a set of queries that were performed for this application; this set consists of 168,352 queries issued between Feb 1, 2016 and Feb 25, 2016. The queries are all aggregations that calculate sums of various measures while grouping by timestamp (hour-of-day). The number of groups in a typical query is quite small, varying between 1 and 12 in most cases.

**Performance:** We first evaluated Seabed's performance on this dataset. We pick a set of 15 queries: five queries each for groups of size 1, 4, and 8. We ran each query ten times, and we calculated the median response time per query. All experiments were run with 100 cores.

Figure 2.14(a) shows the cumulative distribution function of response times for NoEnc, Seabed and Paillier. Seabed's response time ranges from 1.08 to 1.45 times

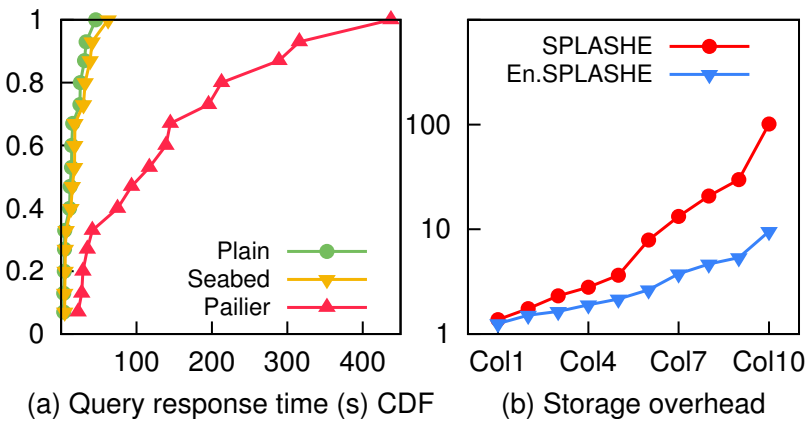



Figure 2.14: Seabed on the Ad Analytics workload: (a) query response-time CDF and (b) storage overhead due to SPLASHE.

that of NoEnc. The median response time for Seabed is 17.8s, whereas for NoEnc it is 13.8s. Thus Seabed's response time is only 27% higher than NoEnc's. On the other hand, the median response time for Paillier is 6.7× that of Seabed. To understand this result in more detail, we looked at the characteristics of the query responses. The average number of rows aggregated for a query across all groups was 210 million, the average size of the ID list was only 163.5KB, and the average number of AES operations required for decryption was roughly 26,000. This shows that there is a lot of contiguity of IDs in the ASHE ciphertext lists. Therefore, while queries could theoretically choose rows at random and thus create huge ID lists, our real-world dataset shows that this does not necessarily happen in practice: the data is stored in a certain order, and Seabed benefits from that order.

In all our experiments, the Seabed client used a high-bandwidth link to connect to the server. To measure the effect of lower-bandwidth and higher-latency links, we artificially changed the network bandwidth/latency between server and client to 100Mbps/10ms and 10Mbps/100ms. This increased the median response time by only 1% in the former case and 12% in the latter case, as the ID lists that need to be transferred are quite small.

**Storage:** We also used this dataset to quantify SPLASHE's overall storage overhead. Through conversations with operators, we determined that 10 out of 33 dimensions

and 10 out of 18 measures require encryption. We used the procedure outlined in Section 2.3.4 to calculate the storage overhead for these 10 dimensions.

Figure 2.14(b) shows the cumulative storage overhead for each of the 10 dimensions in our dataset, sorted by the number of unique values in the dimension. The graph shows that if we restrict the storage overhead to a factor of two, we can encrypt only one dimension with Basic SPLASHE, whereas we can encrypt two dimensions with Enhanced SPLASHE. With a storage overhead of three, we can encrypt only three dimensions with Basic SPLASHE, whereas we can encrypt 6 with Enhanced SPLASHE. In this case, roughly 92% of all queries involve at least one column that uses enhanced SPLASHE.

### 2.6.7 AmpLab Big Data Benchmark

The AmpLab benchmark includes four types of queries (scan, aggregation, join and external script). Some of them come in different variants based on the result/join size, so there are ten queries in total. For this experiment we used 32 cores and loaded the entire Big Data Benchmark dataset (table 2.5) into the workers' memory. We measured the time to perform the query and store the results back into cache memory. Since the Big Data Benchmark is not designed for interactive queries, most of the result sets are huge and cannot fit into one machine's memory. Hence, for this section we do not measure the client-side cost of any of the compared systems.

We had to make a few simplifications to the query set in order to support it. Queries 2 and 4 require substring-search over a column and a text file, respectively. Existing searchable encryption techniques do not efficiently support this operation. Hence we simplified query 2 by matching over deterministically encrypted prefixes, and we simplified query 4 by keeping the text file as plaintext. Query 3 involves sorting based on aggregated values; since this can only be done on the client, and given that we measured only server-side overhead in this experiment, we omitted the sorting step.

Figures 2.12 and 2.13 show the results. Query 1 does not use group-by or aggregation, so all tested systems had much faster response times. Both Seabed and Paillier were slower than NoEnc because of OPE overheads. On the remaining queries Seabed was consistently faster than Paillier, though not as much as we had shown in Sections 2.6.2 and with the Ad Analytics workload. This is because the queries results contained millions of groups and, as we saw in Section 2.6.5, Seabed is slower on result sets with a very small or a very large number of groups. Nevertheless, the results show that Seabed is better than Paillier even for these workloads and is close to NoEnc performance for most queries.

## 2.7 Related Work

**Homomorphic Encryption.** Homomorphic encryption allows computations to be performed on encrypted data such that the computed result, when decrypted, matches the result of the equivalent computation performed on unencrypted data. The first construction of a fully homomorphic scheme that allows arbitrary computations on encrypted data was shown in [71]. However, fully homomorphic schemes are far from practical even today. For example, the amortized cost of performing AES encryption homomorphically is about 2s [74] but this is still $10^8$ times slower than AES over plain text (Section 2.4).

There are also partially homomorphic schemes that allow selected computations on encrypted data. For example, Paillier [132] allows addition of encrypted data while BGN [37] supports one multiplication and several additions. However, these schemes incur significant cost in terms of both computation and storage space. Algorithms to reduce storage overhead by packing multiple integer values into a single Paillier encrypted value are proposed in [70] and implemented in [157].

**Encrypted databases.** CryptDB [138] leverages partially homomorphic encryption schemes to support SQL queries efficiently over encrypted data, and Monomi [138] introduced a split client-server computation model to extend support for most of

the TPC-H queries over encrypted data. However, as we show in this Chapter, the partially homomorphic encryption schemes used in CryptDB and Monomi are not efficient enough to support interactive queries when applied to large datasets.

**Trusted hardware.** Hardware support for trusted computing primitives, such as Intel SGX [116], secure co-processors [87], and FPGA-based solutions [14], are available today. These solutions allow client software to execute in the cloud without providing visibility of client data to the cloud OS. Several prior systems – such as Cipherbase [14], TrustedDB [18], M2R [55] and VC3 [147] – rely on secure trusted hardware to provide privacy-preserving database or MapReduce operations in the cloud.

The use of trusted hardware has the potential to provide secure computations at minimal performance overhead. However the client has to trust that the hardware is free of errors, bugs, or backdoors. It is difficult to confirm that this is indeed the case, since errors can be introduced in both the design of the hardware and in the fabrication process, which is frequently outsourced [88]. In fact, hardware backdoors have been found in real-world military-grade hardware chips [151], and hardware trojan detection is an active research field in the hardware community [32]. We believe that it is useful to develop alternatives that rely only on cryptographic primitives.

**Frequency attacks on property-preserving encryption.** Property-preserving encryption schemes by definition leak a particular property of the encrypted data. For example, deterministic encryption [28] leaks whether two ciphertexts are equal, and order-preserving encryption [36] leaks the order between the ciphertexts. Naveed et al. [125] used auxiliary information and frequency analysis to show that one can infer the plain text from ciphertexts that have been encrypted using such property-preserving encryption schemes.

## 2.8 Conclusion

We have described Seabed, a system for performing Big Data Analytics over Encrypted Data. We have introduced two novel encryption schemes: ASHE for fast aggregations over encrypted data, and SPLASHE to protect against frequency attacks. Our evaluation on real-world datasets shows that ASHE is about an order of magnitude faster than existing techniques, and that its overhead compared to a plaintext system is within 45%.

# 3

# Queries under the non-collusion assumption

## 3.1 Introduction

In the age of "big data", it is well known that many interesting things can be learned by collecting and analyzing large graphs, and a number of tools – including GraphLab [110], PowerGraph [78], and GraphX [79] – have been developed to make such analyses fast and convenient. Typically, these tools assume that the user has a *property graph* $G$ (that is, a graph that has some data associated with its vertexes and/or edges) and wishes to compute some function $F(G)$ over this graph and its properties. A common assumption is that there is a single entity that knows the entire graph $G$ and is therefore able to compute $F(G)$ directly.

However, there is another class of use cases where the graph $G$ *contains sensitive information and is spread across multiple administrative domains*. In this situation, each domain knows only a subset of the vertexes and edges, so it cannot compute $F(G)$ on its own, but the domains may not be willing to share their data with each other because of privacy concerns.

One interesting real-world instance of this problem is the computation of *systemic risk* in financial networks [8]. Motivated in part by the financial crisis of 2008, this topic has recently seen a lot of interest in the theoretical economics literature. Briefly, economists have discovered that one of the causes for the crisis was a "snowball effect" in which a few initial bankruptcies caused the failure of more and more other banks due to financial dependencies. *In theory*, it would be possible to quantify the risk of such a cascading failure by looking at the graph of financial dependencies between the banks, and in fact economists have already developed a number of metrics [61, 63] that could be used to quantify this "systemic" risk, and to ideally give some early warning of an impending crisis.

However, *in practice*, the required information is extremely sensitive because it directly reflects the business strategy of each bank. It is so sensitive, in fact, that banks would prefer not to share it even with the government. This is why current audits (such as the annual "stress tests" that were introduced after the crisis, e.g., by the Dodd-Frank Act in the United States) are strictly compartmentalized, so that each auditor is only allowed to look at the data of one particular bank. This provides some basic security, but it is not sufficient to discover complex interdependencies, which would require looking at data from *all* the banks. This is why, in a recent working paper [66], the Office of Financial Research (OFR) has started investigating ways to perform system-wide stress tests while protecting confidentiality.

One possible approach would be to use secure multiparty computation (MPC) [163], which would enable the banks to collectively evaluate a function $F(G)$ over their combined financial data – say, one of the existing systemic risk measures [61, 63]. However, there are two challenges with this approach. The first is performance: MPC does not scale well to large numbers of parties or complex computations. As we will show, computing systemic risk with a straightforward application of MPC would literally take many years.

The second, and somewhat more subtle, challenge is privacy: MPC only guar-

antees that no one can learn the *intermediate* results of the computation. However, even the *final* result (the value of $F(G)$) can reveal information about the underlying graph $G$, especially in the presence of auxiliary information. To see this intuitively, imagine using MPC to compute the average weight of the people in a room. This will not reveal the weight of any single individual, but the adversary can still infer the presence of a team of sumo wrestlers by looking only at the final result. A similar concern arises in the context of financial data [66, §4.2].

In this Chapter, we present a system called DStress that can efficiently analyze graphs that are spread across thousands of administrative domains, while giving strong, provable privacy guarantees on *both* the topology of the graph $G$ *and* the data it contains. DStress supports *vertex programs*, a programming model that is also used in Pregel [112] and Graphlab [110], two popular frameworks for non-confidential graph computations. It addresses the first challenge with a special graph-computation runtime that can execute vertex programs in a distributed fashion, using MPC and a variant of ElGamal encryption for transferring data between domains, and it addresses the second challenge by keeping intermediate results encrypted at all times, and by offering differential privacy [60, 58] on the final result.

We have built a prototype of DStress, and we have evaluated it using two systemic-risk models from the theoretical economics literature. Our results show that these models could be evaluated on the entire U.S. banking system in less than five hours on commodity hardware, using about 750 MB of traffic per bank. In Appendix B.3, we also show that the use of differential privacy (which was already suggested by the OFR working paper [66]) does not significantly diminish the utility of the systemic risk measure. In summary, this Chapter makes the following three contributions:

- DStress, a scalable system for graph analytics with strong privacy guarantees (Section 3.3);
- an application of DStress to privately measuring systemic risk in financial net-

works (Section 3.4); and

- an experimental evaluation, based on a prototype implementation of DStress (Section 3.5).

## 3.2 Overview

We consider a scenario with a group of $N$ participants $P_i$, $i = 1, \ldots, N$ that each know one vertex $v_i$ of a directed graph $G$, as well as a) the edges that begin or end at $v_i$, and b) any properties associated with $v_i$. The participants wish to collectively compute a function $F(G)$, such that:

- **Value privacy:** The computation process does not reveal properties of $v_i$ to participants other than $P_i$;
- **Edge privacy:** The computation process does not reveal the presence or absence of an edge $(v_i, v_j)$ to participants other than $i$ and $j$; and
- **Output privacy:** The final output $F(G)$ does not reveal too much information about individual vertexes or edges in $G$.

In the scenarios we are interested in, the number of parties $N$ is on the order of thousands; for instance, the number of major banks in the U.S. banking system is about $N = 1,750$.

### 3.2.1 Background: Systemic risk

To explain how financial networks fit this model, we now give a very brief introduction to systemic risk. Since this is a complex topic, we focus on the aspects that are most relevant to DStress; for more details, see [39, 66, 33].

Banks and other financial institutions, as part of their regular business with clients, are exposed to *risk*. We can think of this risk as a specific abstract event $x_i$ – such as a drop in house prices – that, if it happened, would cause bank $b$ to lose a certain amount of money $y_i$. Thus, $b$'s *balance sheet* has *exposure* of the form: `(if` $x_i$

`then` $-y_i$`)`. To prevent a buildup of excess exposure to any single future event, $b$ can in advance create *derivatives* on event $x_i$ and sell part of this exposure to other banks (presumably for a fee). We can think of these derivatives as "insurance contracts" that specify that a certain sum will be due if and when a particular event occurs. Thus, if $b$ bought "insurance" against $x_i$ from another bank that pays $z_i$, $b$'s exposure would now be: `(if` $x_i$ `then` $z_i - y_i$`)`. More complicated forms of derivatives also exist.

Banks regularly reinsure their risk by buying additional derivatives from other banks. The result is a network of dependencies that spans the entire financial sector. We can think of this as a graph $G$ that contains a vertex for each bank and an edge $(b_1, b_2)$ whenever $b_1$ has sold a derivative to $b_2$. Vertexes would be annotated with the liquid cash reserves of the corresponding bank, and edges would be annotated with the payment that is due in each event. Using this graph, it is possible to essentially simulate what would happen if a particular event were to occur – including possible cascading failures, where the initial bankruptcy of a few critical banks causes a "domino effect" that eventually affects a large fraction of the network. The expected "damage" (and thus the systemic risk) can then be measured in a variety of ways – e.g., as the amount of money the government would need to inject in order to stabilize the system. In Section 3.4, we discuss two concrete models from the economics literature in more detail.

### 3.2.2 Strawman solutions

One obvious way to compute the systemic risk would be to create an all-powerful government regulator that has access to the financial data of all the banks. However, this does not seem practical, since banks critical rely on secrecy to protect their business practices [8]. Currently, regulatory bodies that deal with *less* sensitive information about *individual* banks already have extremely restrictive legal safeguards and multiple levels of oversight [66, §3.1].

Another potential approach, first suggested by [8], would be to use multi-party computation (MPC) [163]: one could design a circuit that takes each bank's books as inputs, executes the simulation in MPC, and finally outputs the desired measure of risk. This approach would be more palatable for the banks, since they would not need to reveal their secret inputs. However, the circuit would be enormous: even the simplest models of contagion in the literature essentially require raising a $N \times N$ matrix to a large power, where $N$ is the number of banks (i.e., about $1,750$). Despite recent advances in MPC, such as [35, 43, 47, 166], evaluating such a large circuit with $N = 1,750$ parties is far beyond current technology.

The cost of MPC could be reduced somewhat by delegating the computation to a smaller number of parties, as in Sharemind [35] or PICCO [166]. However, this approach would do nothing to reduce the size of the circuit, and, given the high stakes involved, the number of parties would still need to be large – the largest collusion case reported in the literature involved 16 banks [144]!

As discussed earlier, none of these approaches would provide output privacy, and this would be a serious concern, since the final output (i.e., the current level of systemic risk) could be enough for some of the banks to make inferences about the graph, particularly if they already know some of the other edges and vertexes.

### 3.2.3 Our approach

Our approach is based on two key insights. Our first observation is that much of the enormous cost of the MPC-based strawman comes from the fact that the graph is itself confidential and therefore must be an input to the computation. We can get around this by formulating the function $F$ as a *vertex program* – that is, as a sequence of computations at each vertex that are interleaved with message exchanges over the edges – and by executing it in a distributed fashion. This is safe because each participant already knows the edges that are adjacent to her vertex; the main challenge is to prevent information leakage through intermediate results. In DStress,

we accomplish this with a combination of secret sharing, small MPC invocations for the computations at each vertex, and a special protocol for transferring shares without revealing the topology of the graph.

Our second key insight is that we can use *differential privacy* [60] to achieve output privacy. Differential privacy provides strong, provable privacy guarantees, which should be reassuring to the banks. Its main cost is the addition of a small amount of random noise to the output, but, since we are looking for early warnings of large problems, a bit of imprecision (e.g., a shortfall of $1 billion is reported as $0.95 billion) should not affect the utility of the results. If a potential problem is detected, a more detailed investigation could be conducted outside of our system.

## 3.3 The DStress system

We begin by briefly reviewing three technologies that DStress relies on: differential privacy, secure multiparty computation, and ElGamal encryption.

**Differential privacy:** DStress is designed to provide *differential privacy* [60], one of the strongest known privacy guarantees. Differential privacy has a number of features that are attractive in our setting, such as protection against attacks based on auxiliary data (which have been the source of several recent privacy breaches [20, 123, 26]), strong composition theorems, and a solid mathematical foundation with provable guarantees.

Differential privacy is a property of *randomized* queries – i.e., the query computes not a single value but rather a probability distribution over the range $R$ of possible outputs, and the actual output is then drawn from that distribution. This can be thought of as adding a small amount of noise to the output. Intuitively, a query is differentially private if a small change to the input only has a statistically negligible effect on the output distribution.

More formally, let $I$ be the set of possible input data sets. We say that two input data sets $d_1, d_2 \in I$ are similar (and we write $d_1 \sim d_2$) if they differ in at most one

element. Then, a randomized query $q$ with range $R$ is $\varepsilon$-differentially private if, for all possible sets of outputs $S \subseteq R$ and all input data sets $d_1, d_2 \in I$ with $d_1 \sim d_2$,

$$Pr[q(d_1) \in S] \leq e^{\varepsilon} \cdot Pr[q(d_2) \in S].$$

That is, any change to an individual element of the input data can cause at most a small multiplicative difference ($e^{\varepsilon}$) in the probability of *any* set of outcomes $S$. The parameter $\varepsilon$ controls the strength of the privacy guarantee; smaller values result in better privacy. For more information on how to choose $\varepsilon$, see, e.g., [85].

A common way to achieve differential privacy for queries with numeric outputs is the *Laplace mechanism* [60], which works as follows. Suppose $\bar{q} : I \to R$ is a deterministic, real-valued function over the input data, and suppose $\bar{q}$ has a finite *sensitivity* $s$ to changes in its input, i.e., $|\bar{q}(d_1) - \bar{q}(d_2)| \leq s$ for all similar databases $d_1, d_2 \in I$. Then $q := \bar{q} + Lap(s/\varepsilon)$, i.e., the combination of $\bar{q}$ and a noise term drawn from a Laplace distribution with parameter $s/\varepsilon$, is $\varepsilon$-differentially private. This corresponds to the intuition that the more sensitive the query, and the stronger the desired guarantee, the more "noise" is needed to achieve that guarantee.

**Secure multiparty computation:** DStress relies on secure multiparty computation (MPC) to perform certain steps of the graph algorithm it is running. MPC is a way for a set of mutually distrustful parties to evaluate a function $f$ over some confidential input data $x$, such that no party can learn anything about $x$ other than what the output $y := f(x)$ already implies. In the specific protocol we use (GMW [77]), each party $i$ initially holds a *share* $x_i$ of the input $x$ such that $x = \oplus_i x_i$ (in other words, the input can be obtained by XORing all the shares together), and, after the protocol terminates, each party similarly holds a share $y_i$ of the output $y = f(x)$. The function $f$ itself is represented as a Boolean circuit. GMW is *collusion-resistant* in the sense that, if $k+1$ parties participate in the protocol, the confidentiality of $x$ is protected as long as no more than $k$ of the parties collude.

**ElGamal encryption:** For reason that will become clear later, DStress requires an

encryption scheme with two unusual properties: an additive homomorphism and a way to re-randomize public keys. Both can be elegantly accomplished using a variant of ElGamal [62]. The original ElGamal scheme consists of three functions: a key generator, an encryption and a decryption function. These functions are defined over some cyclic group $G$. Assume $G$ is of order $q$ and has a generator $g \in G$. ElGamal's key generator function returns a random element $x \in Z_q$ as the secret key, and a public key $h = g^x$. Moreover, given a public key $h$ and a message $m$, the encryption function picks a random $y \in Z_q$ (sometimes called an ephemeral key) and returns the ciphertext $c = (g^y, m \cdot h^y) = (g^y, m \cdot g^{xy})$. Given a ciphertext $c = (c_1, c_2)$, and a secret key $x$, the decryption function first computes $s = c_1^x = g^{xy}$, then $s^{-1} = c_1^{(q-x)} = g^{y(q-x)}$ and finally returns the recovered plaintext as $c_2 \cdot s^{-1} = mg^{xy}g^{qy-xy} = mg^{qy} = m$.

ElGamal itself has a multiplicative homomorphism: if we encrypt two messages $m_1$ and $m_2$ and multiply the two ciphertexts together, the result decrypts to the product $m_1 \cdot m_2$. However, this can be turned into an additive homomorphism using a small trick, which is to encrypt not the message $m$ itself but rather $g^m$. The resulting scheme (exponential ElGamal [50]) ensures that the product of two ciphertexts $g^{m_1} \cdot g^{m_2} = g^{m_1+m_2}$ now decrypts to the sum of the underlying messages. The downside is that there is no easy way to go back from $g^m$ to $m$ – but, if the number of valid messages is small enough, the recipient can use a lookup table to decrypt: simply precompute $g^c$ for all candidate messages $c$ and compare the results to the $g^m$ she received. Exponential ElGamal also satisfies our second requirement: if $g^x$ is a public key, we can re-randomize it by raising it to some value $r$, yielding a new public key $g^{xr}$. If a message is encrypted with this new public key, it will not decrypt with the original private key $x$; however, this can be fixed by raising the ephemeral key in the ciphertext to $r$ as well. Notice that both operations (re-randomizing the public key and adjusting ciphertexts) can be performed without knowledge of the private key $x$.

### 3.3.1 Programming model

DStress is designed to run *vertex programs*. A vertex program consists of (1) a graph $G := (V, E)$; (2) for each vertex $v \in V$, an initial state $s_v^0$ and an update function $f_v$; (3) a number of iterations $n$; (4) an aggregation function $A$; (5) a no-op message $\perp$; and (6) a sensitivity $s$. DStress executes such an algorithm as follows. First, each vertex $v$ is first set to its initial state $s_v^0$. Next, DStress performs a *computation step* by invoking the update function $f_v$ for each vertex, which outputs a new state $s_v^1$ and, for each neighbor of $v$ in $G$, exactly one message. (If more messages need to be sent, they can be included in one larger message.) When $v$ has no data to send to some neighbor, it outputs the no-op message $\perp$ instead; this is necessary to avoid leaking information through its communication pattern. The computation step is followed by a *communication step*, in which each vertex sends its messages along the edges to its neighbors; the recipients then use the messages as additional inputs for their next computation step. After $n$ computation and communication steps, DStress performs a final computation step and then invokes the aggregation function $A$, which reads the final state of each node and combines the states into a single output value. Finally, DStress draws a noise term from a Laplace distribution $Lap(s/\varepsilon)$ and adds it to the output value, which yields result of the computation.

The vertex programming model is quite general; there are other systems that implement it – such as Pregel [112] – and it can express a wide variety of graph algorithms. Not all of these algorithms have privacy constraints, but there are many that do; for instance, cloud reliability [165], criminal intelligence [153, 100], and social science [104, 40, 64] all involve analyzing graphs that can span multiple administrative domains.

### 3.3.2 Threat model and assumptions

DStress relies on the following five assumptions:

1. The nodes are honest but curious (HbC), i.e., they will faithfully execute

DStress but try to learn as much about the graph as they can;

2. The nodes do not have enough computational power to break the cryptographic primitives we use;
3. There is an upper bound $k$ on the number of nodes that will collude;
4. There is a (publicly known) upper bound $D$ on the degree of any node in the graph; and
5. There is a trusted party (TP) that knows the identities of all the nodes in the system and can perform some simple setup steps. (The TP can be offline and never sees any private information.)

Assumption 1 may seem counterintuitive at first, especially in our systemic-risk case study, which involves valuable financial data. However, recall that banks are already heavily audited and inspected in most countries. These audits are compartmentalized, so they cannot be used to measure systemic risk directly; however, they could certainly be used to verify that each bank has input the correct data and has executed DStress correctly.

Assumption 2 is standard for virtually all protocols that use cryptography; it implies that DStress offers *computational* differential privacy [120]. Assumption 3 seems plausible for the banking scenario because of antitrust laws that prevent large-scale collusion between banks; for other applications, the bound $k$ could be chosen based on the largest observed instance of collusion, plus a safety margin. Assumption 4 is in accordance to economic incentives described in [48], which suggest that the financial network is not fully connected. An example of an institution that satisfies assumption 5 is the Federal Reserve.

### 3.3.3 Basic operation

When executing an algorithm, DStress runs on a distributed set of *nodes*, and it maps each vertex in the graph to a specific node that will provide the initial state for that vertex and that will coordinate the corresponding computation and communication

steps. Unlike the (logical) vertexes, which communicate over edges in the graph, the (physical) nodes can communicate directly over a network, such as the Internet. We expect that, for privacy reasons, each participant would want to operate its own node, so that it does not have to reveal its initial state to another party.

To prevent privacy leaks, DStress must not allow any node to see intermediate states of the computation – not even that of their own vertex, since it may have changed based on messages from other vertices. Hence, DStress associates each node $i$ with a set $B_i$ of other nodes that we refer to as the *block* of $i$. The members of the block each hold a share of the vertex's current state, and they use MPC to update the state based on incoming messages. To prevent colluding nodes from learning the state of a vertex, each block contains $k+1$ nodes, where $k$ is the collusion bound we have assumed.

A key challenge is to enable vertices to communicate without weakening security or revealing the structure of the graph. If $(i, j)$ is an edge in the graph and $i$ wants to send a message $m$ to $j$, then the members of block $B_i$, who would each hold a share of $m$ after the MPC that generated it, cannot simply send their shares of $m$ to the members of block $B_j$, since that would reveal the existence of the edge $(i, j)$ to both blocks. To avoid this problem, DStress redirects all communication between blocks $B_i$ and $B_j$ through the nodes $i$ and $j$, who already know that an edge exists between them. To prevent $i$ and $j$ from reconstructing $m$, all shares are encrypted, and we use ElGamal's key re-randomization feature to prevent the senders from learning the identities of the recipients through their public keys.

### 3.3.4 One-time setup step

Before DStress can be used with a new graph, it is necessary to perform a one-time setup step. This step has two different purposes. First, it associates each node $i$ with a block $B_i$ that contains $k+1$ *different* nodes, including $i$. This is necessary to prevent curious nodes from filling their own blocks with Sybil identities or with mul-

tiple instances of the same node, which would weaken DStress's collusion-resistance. Second, it equips each block $B_i$ with $D$ different sets of public keys. This is necessary to prevent colluding neighbors of $i$ from identifying members of $B_i$ based on their public keys.

The setup step is coordinated by the trusted party (TP). The TP begins by asking each node $i$ for a) $i$'s public ElGamal key, and b) $D$ different *neighbor keys* $n_1^i, \ldots, n_D^i$, which $i$ can choose arbitrarily from $Z_q$. The TP then randomly picks a list of members for $i$'s block $B_i$ and publishes $\sigma_{\text{TP}}(\{(k, B_k)\}_{k=1..|V|}, B_A)$ – that is, a list of nodes and their blocks that is signed with the TP's private key. (Note that this list does not contain information about edges and thus reveals nothing about the structure of the graph.) $B_A$ is a special block that is used for aggregation (Section 3.3.6); its $k+1$ members are also chosen randomly by the TP.

Next, the TP generates $D$ *block certificates* for each block $B_i$. A block certificate is a tuple $C_{i,j} := \sigma_{\text{TP}}(g^{x_1 n_j^i}, g^{x_2 n_j^i}, \ldots, g^{x_D n^i, j})$, that is, it contains one public key for each member of the corresponding block, but the keys in the $j$.th certificate for node $i$ are re-randomized using the $j$.th neighbor key from node $i$. The TP signs each of the $D$ block certificates and then sends them to node $i$, who forwards each certificate to a different neighbor. (If $i$ has fewer than $D$ neighbors in the graph, it simply discards the leftover certificates.) Finally, each node $i$ distributes the block certificates it has received from its neighbors to the members of its own block $B_i$, but without identifying the specific neighbor – $i$ only tells the members of $B_i$ which certificate is for its first neighbor, its second neighbor, and so on.

Once this step is completed, the TP is no longer needed and can leave the system. Notice that the TP has never learned the topology of the graph.

### 3.3.5 Message transfer protocol

Next, we describe how the block certificates can be used to securely send messages along edges of the graph. Since the details of the full protocol are somewhat com-

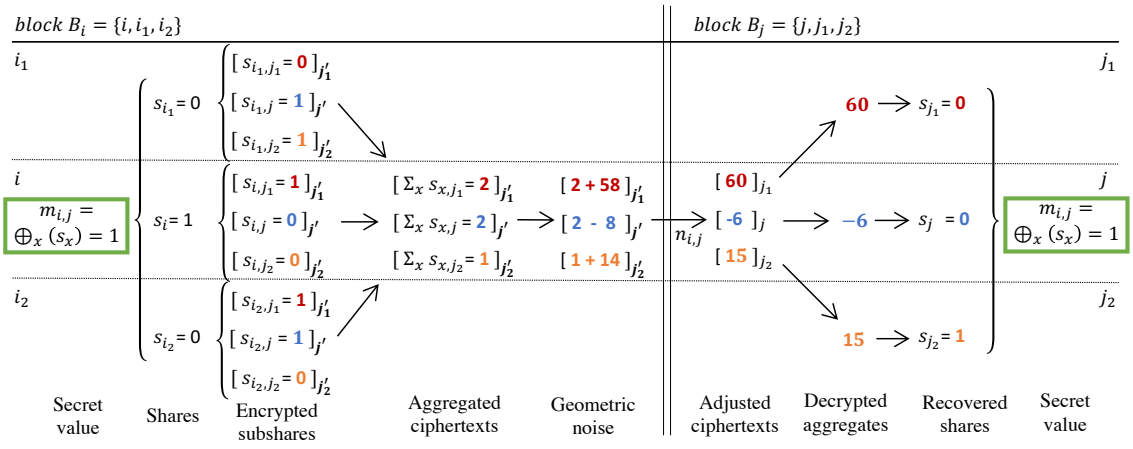



Figure 3.1: A message transfer example between two blocks of three nodes. $[\ldots]_{j'_x}$ denotes a ciphertext encrypted with the randomized public key of $j_x$, and $[\ldots]_{j_x}$ denotes a ciphertext encrypted with the original key.

plicated, we start with a simple but flawed strawman protocol and then derive the full protocol in several steps.

Recall that, at the end of each computation step, each node $i$ sends a message $m_{i,j}$ (possibly the no-op message $\perp$) to each neighbor $j$ in the graph. Since the computation step is performed in MPC, at the end of the step each member of $i$'s block $B_i$ holds one share of $m_{i,j}$, such that the message can be reconstructed by XORing all the shares together. These shares must be transferred to the members of $B_j$, who then use them as inputs to $j$'s' next computation step.

***Strawman #1:*** *Each $x \in B_i$ picks a different public key from $j$'s block certificate and encrypts its share $s_x$ of $m_{i,j}$ with this key. Then the members of $B_i$ forward their encrypted shares to $i$, who forwards them to $j$. $j$ adjusts the ephemeral keys in the ciphertexts using the neighbor key $n_{i,j}$ and then forwards them to the members of $B_j$, who decrypt them and thus each obtain one share of $m_{i,j}$.*

This approach prevents the members of $B_i$ and $B_j$ from learning about each other directly: all communication is via $i$ and $j$, and the members of $B_i$ only see the re-randomized public keys of the members of $B_j$, so they cannot identify the latter by recognizing their public keys. However, this approach weakens collusion resistance: if the same node $n$ happens to be a member of both $B_i$ and $B_j$, or if two nodes $n_1 \in B_i$

and $n_2 \in B_j$ collude, they can potentially learn *two* shares. To prevent this, we make the following change:

***Strawman #2:*** *Like strawman #1, except that each $x \in B_i$ splits its share $s_x$ into $k+1$* subshares $s_{x,1}, \ldots, s_{x,k+1}$ *such that $s_x = \oplus_{y=1..k+1} s_{x,y}$ and then encrypts a different subshare for each member of $B_j$.*

As long as the secret-sharing scheme is associative and commutative, the members of $B_j$ can obtain valid (but different) shares of $m_{i,j}$ simply by combining all the subshares they receive. This change also restores collusion resistance: as long as both $B_i$ and $B_j$ have at least one member that does not collude, the colluding nodes will always miss at least one share, namely the one that is sent between the two non-colluding nodes. However, if $n_1 \in B_i$ and $n_2 \in B_j$ collude, they can still infer the presence of edges by recognizing subshares: $n_1$ can use some external channel to tell $n_2$ about the subshares it has sent, and if $n_2$ subsequently receives one of them, they can infer that their blocks are connected by an edge. We fix this as follows:

***Strawman #3:*** *Like strawman #2, except that each member of $B_i$ breaks its subshare into individual bits and encrypts each bit separately. The encrypted bits are forwarded through i as before, but, rather than forwarding them to j directly, i uses the homomorphic addition in exponential ElGamal to combine the corresponding bits from different subshares. This yields the encrypted sum of bits of the shares, which j then forwards to $B_j$. Members of $B_j$ decrypt the sums and set their bit share to 0 iff the sum is even.*

This approach *almost* meets our requirements, since the recipients never see the senders' original subshares and thus cannot recognize them. However, because the homomorphic operation is an addition and not an XOR, the encrypted "bits" that arrive at $B_j$ are actually numbers that correspond to the number of ones in the original subshares. This leaks some information about the original shares. To see why, consider the (extreme) case where the adversary controls $k$ of the $k+1$ nodes in both $B_i$ and $B_j$, and wishes to learn whether the edge $(i, j)$ exists. Suppose that, during some particular communication step, the subshares of the adversary's nodes in $B_i$ for

each of the $D$ messages add up to $S_1, \ldots, S_D$, respectively. If the adversary's nodes in $B_j$ then receive only sets of shares that add up to less than $S_n - 1$ or more than $S_n + 1$ for all $1 \leq n \leq D$, then the adversary knows that none of the received messages could have come from $B_i$, so the edge $(i, j)$ cannot exist. Conversely, if, over the course of many communication steps, the adversary's nodes in $B_j$ always receive some set of shares that add up to $S_n \pm 1$ for some $n$, the adversary can be increasingly confident (though never completely certain) that the edge $(i, j)$ does exist. If the adversary controls fewer than $k$ of the $k+1$ nodes in $B_i$ and/or $B_j$, the risk of the adversary learning something about $(i, j)$ in any particular communication step diminishes, but it never completely disappears. We mitigate this risk by making one final change:

***Final protocol:*** *Like strawman #3, except that i homomorphically adds an even random number from* $2 \cdot Geo(\alpha^{\frac{2}{k+1}})$ *to each encrypted bit before forwarding it to* $B_j$ *via* $j$*, where Geo is the geometric distribution*[1] *as described in [75] and* $\alpha$ *is a parameter in* $(0, 1)$.

This preserves correctness: the recipients will receive an even (but otherwise random) number if and only if it would have received a zero bit using strawman #3. However, the adversary's chances of learning something useful have diminished dramatically: the sum of bits is now noised, and it is very hard for the adversary to extract information from this side-channel. In fact, as [75] shows, the application of geometric noise provides $\varepsilon$-differential privacy, where $\varepsilon = -\ln \alpha$. This way, we can maintain a privacy budget to keep track of what the adversary learns and make sure that the probability of an edge leaking is minimal. For details regarding the sensitivity analysis and differential privacy guarantees of the protocol, please refer to Appendix B.2.

The overall effect is that, for each edge $(i, j)$ in the graph, the members of block $B_i$ can transfer the shares of a message to the members of block $B_j$ such that a) no group of $k$ or fewer colluding nodes can learn the contents of the message, and that b) edge privacy (Section 3.2) is maintained. Appendix B.1 includes a formal proof

[1] The geometric distribution is a discretized version of the Laplace distribution, which is widely used in differential privacy.

of the first property and Appendix B.2 a detailed discussion of edge privacy.

### 3.3.6 Executing a program

Next, we describe how DStress executes a program. For simplicity, we focus only on the algorithm and ignore practical challenges, such as fault tolerance; these challenges are orthogonal and can be addressed with existing techniques. Recall that each execution has $n$ computation and communication steps, followed by aggregation and noising.

**Initialization step.** DStress maintains the invariant that, at the beginning of each computation step, each member of a node $i$'s block $B_i$ has 1) a share of the current state of $i$'s vertex, and 2) shares of $D$ input messages, which can either be messages of $i$'s neighbors or instances of the no-op message $\perp$. To make this invariant true at the first step, each node $i$ starts by loading the initial state of its local vertex, as well as $D$ copies of $\perp$ (since there are no real messages yet), and splits each of them into $|B_i|$ shares, one for each member of its block $B_i$.

**Computation step.** In each computation step, the members of each block $B_i$ use MPC to evaluate the update function of the corresponding vertex $v_i$. The circuit has inputs for the $D$ input messages and the current state of $v_i$, as well as outputs for $D$ output messages and the new state of $v_i$. (If the degree of the vertex is less than $D$, some of the messages are copies of $\perp$.) Note that both inputs and outputs of an MPC step remain shared among the members of the block and are never revealed to any individual node.

**Communication step.** In this step, DStress invokes the protocol from Section 3.3.5 to send each message along the corresponding edge of the graph. Because each directed edge is used to send exactly one message, a node can immediately proceed to the next computation step once it has received a message from each of its in-neighbors; there is no need for global coordination.

**Aggregation+noising step.** Once $n$ computation and communication steps have

been performed, each block $B_i$ holds shares of vertex $v_i$'s final state. Next, DStress evaluates the aggregation function $A$ on the final states, using the special aggregation block $B_A$. Each block sends its state shares and some random shares to $B_A$; the members of $B_A$ then use MPC to a) evaluate $A$ on the states; b) combine the random shares to get a random input seed, c) draw a noise term from $\text{Lap}(s/\varepsilon)$ using the seed; and d) output the sum of that term and the result of $A$. The simplest way to implement this is to use a single aggregation block, but this could become a bottleneck for larger graphs; in this case, the aggregation can be performed hierarchically, using a tree of aggregation blocks.

### 3.3.7 Limitations

DStress currently executes a fixed number of iterations. Dynamic convergence checks are problematic from the perspective of differential privacy because the number of rounds is itself disclosive and would need to be treated as an additional output. However, if the number of rounds is chosen conservatively, this restriction will cost some performance but should not affect correctness.

DStress is limited to executing vertex programs that a) can be expressed as Boolean circuits, and b) have a known, finite sensitivity bound. The first limitation exists because DStress uses MPC to execute the computation steps; it effectively means that the update functions cannot have dynamic loop bounds or unbounded recursion. The second limitation currently prevents ad-hoc queries, but we speculate that DStress could be augmented with automated sensitivity inference, e.g., using linear type systems [82, 67] or a system like CertiPriv [21]. Sensitivity inference for graphs is considered challenging, but the community is making progress with systems like wPINQ [139], which can automatically derive the sensitivity for an important class of graph algorithms. Also, one can find algorithms with known sensitivity in the differential privacy literature (e.g., in [96]), as we did for the two algorithms we used in section 3.4.4.

DStress's current design assumes a single bound $D$ on the degree of each vertex in the financial network. If, despite the evidence in [48], the maximum degree was very large, this would slow down our algorithm. However, one could avoid this by dividing the vertexes into buckets based on their approximate degree – e.g., one bucket for vertexes with fewer than 100 neighbors and another for the rest. This would reveal a small amount of information about the degree of each bank (which would probably be heavily correlated with the bank's size), but in return, the MPC block computations for most banks would be much faster than if a single conservative degree bound were used for all banks.

## 3.4 Case studies

In this section, we describe two different models of financial contagion from the economics literature, and we show how they can be implemented in DStress to compute a measure of systemic risk.

### 3.4.1 Metrics and Privacy Guarantees

We follow a recommendation from the OFR working paper [66, §4.3] and measure systemic risk as the *total dollar shortfall (TDS)* – that is, the amount of extra money that the government would need to make available to prevent failures if the contracted event were to occur. It would perhaps be more intuitive to compute the number of banks that fail, but TDS has two key advantages. First, it is more meaningful because it can distinguish between a small shortfall such as $\$10,000$ (which, in a large bank, is easily fixed) and a large, more serious shortfall such as \$10 billion. Second, it is a better fit for differential privacy. It is well known that the answer to many questions about graph-shaped data can change radically when even a single edge is added or removed, and thus such questions cannot be answered with differential privacy. However, the TDS is an exception: adding or removing edges does not disproportionally affect the TDS [84].

The privacy guarantee that results when we add noise to the TDS is called *dollar*-differential privacy; it was first introduced in [66]. In this model, the sensitive data we are protecting consists of the investment portfolios of all the banks, and we consider two data sets $d_1, d_2$ to be similar ($d_1 \sim d_2$) if one can be transformed into the other by reallocating at most $T$ dollars in a single portfolio. This means that an adversary can increase her knowledge about the contribution of financial institutions, whose *aggregate* positions are under $T$ dollars, up to a small multiplicative factor $\varepsilon$.

Note that dollar-differential privacy is different than edge-differential privacy, which is the guarantee provided by the message transfer protocol of DStress. Appendix B.2.1 describes that the overall privacy guarantee of DStress is that adversaries can increase their knowledge about *individual* positions with value up to $T$ by a small multiplicative factor.

### 3.4.2 The Eisenberg-Noe Model

Our first model, from Eisenberg and Noe [61], considers banks holding debt contracts from and to other banks. A stress test based on this model would first, based on some hypothetical future scenario[2], compute a netted exposure graph on a bilateral basis between the banks, as is done in per-bank stress tests today. After computing each bank's contractual obligations, this would result in a graph of payments between the banks. Then, each bank's liquid reserves plus incoming payments (i.e., debts paid by other banks) would be compared to its total debt. If the debts are bigger, the bank would be deemed bankrupt, and its payments would be adjusted based on what assets the bank actually has. As proven in [61], if there are $n$ banks, this process converges to a unique solution after at most $n$ iterations.

Algorithms 1 to 4, show an implementation in DStress. Initially, the algorithm assumes that each node can pay its obligations in full (`prorate=1`); in each update step, each node $i$ computes its local shortfall as a fraction of its debt, and sends a

[2] Regulators choose one or more hypothetical events/shocks, and they build custom models based on those described in the economics literature [65].

**Algorithm 1:** EN-INIT(i)

```
1  begin
2      cash[i] = Liquid reserve at i
3      debt[i][j] = Debt owed by i to j
4      credits[i][j] = Debt owed by j to i
5      totalDebt[i] = ∑_j (debt[i][j])
6      prorate[i] = 1.0
7      noOpMessage = 0
8      sensitivity = 1/r                          // See Section 3.4.4
9  end
```

**Algorithm 2:** EN-UPDATE(i)

```
1  begin
2      liquid = cash[i]
3      foreach j in neighbors(i) do
4          shortfallJ = recvFrom(j)
5          liquid += credits[i][j]-shortfallJ
6      end
7      if liquid < totalDebt[i] then
8          prorate[i] = liquid / totalDebt[i]
9      end
10 end
```

message to each adjacent node $j$ that contains the amount of $i$'s debt to $j$ that $i$ is unable to pay. The final aggregation step computes the TDS.

### 3.4.3 The Elliott-Golub-Jackson Model

The second model, from Elliott, Golub, and Jackson [63], describes a very different type of contagion, using equity cross-holdings to represent inter-institutional dependencies. In this model, there is a set of *primitive assets*, which have associated prices. Banks own their own individual basket of these assets, as well as potentially equity in each other. Thus, the valuation of a bank is a) the value of its own primitive assets, plus b) its fraction of the primitive assets owned by other banks in which it (directly or transitively) holds an equity stake. The latter can be computed via fixpoint itera-

**Algorithm 3:** EN-COMMUNICATE-WITH(i)

```
1 begin
2     foreach j in neighbors(i) do
3         sendTo(j, debts[i][j]*(1-prorate[i]))
4     end
5 end
```

**Algorithm 4:** EN-AGGREGATE(i)

```
1 begin
2     totalShortfall = ∑_i (totalDebt[i]*(1-prorate[i]))
3 end
```

tion. The model has another unusual feature: when a bank's valuation falls below a bank-specific threshold, it is considered to have failed, and its value drops by an additional penalty. This is different from the Eisenberg-Noe model, which is inspired by the allocation of assets in traditional bankruptcy proceedings; the intent is to represent "distressed" institutions that may not fail to the point of actual bankruptcy but still face sudden additional costs due to, e.g., a downgraded credit rating.

[63] also shows that the fixpoint is not unique and depends on the starting conditions and on which nodes fail first; thus, there is a possibility of false negatives. However, this is not due to our implementation in DStress – it is simply how the algorithm works as originally proposed. The algorithm is also *not* guaranteed to converge after $n$ steps, as each step can cause a valuation drop even beyond the discontinuous drop. However, as shown in [84], it converges to its final value monotonically, and thus a limited number of iterations provides a good approximation result.

Algorithms 5 to 8, show an implementation in DStress. Initially, each bank has some exogenous valuation `origVal`; in each step, each bank computes a discount to its own value, based on its primitive assets and the current valuation of its equity holdings, and then propagates that discount to its neighbors in the graph. The final aggregation step computes the TDS of all failed banks relative to their failure threshold, as suggested by [63].

**Algorithm 5:** EGJ-INIT(i)

```
1  begin
2      base[i] = Base assets held by i
3      origVal[i] = initial valuation of i
4      value[i] = current valuation of i
5      insh[i][j] = share of j held by i
6      threshold[i] = i's failure threshold
7      penalty[i] = penalty if < threshold
8      noOpMessage = 0
9      sensitivity = 2/r                          // See Section 3.4.4
10 end
```

**Algorithm 6:** EGJ-UPDATE(i)

```
1  begin
2      value[i] = base[i]
3      foreach j in neighbors(i) do
4          discount = recvFrom(j)
5          value[i] += insh[i][j]*(1-discount)*origVal[i][j]
6      end
7      if value < threshold[i] then
8          value[i] = value[i] - penalty
9      end
10 end
```

### 3.4.4 Sensitivity bounds

Recall that DStress requires the programmer to provide a bound on the program's sensitivity to changes in its input. We rely on a proof by Hemenway and Khanna [84], which shows that the sensitivity of the Elliott-Golub-Jackson algorithm is $2/r$, where $r$ is an upper bound on the leverage ratio of the banks (that is, the ratio between a bank's total assets and its equity may not exceed $1 : r$). This type of constraint is already mandated by law today because leverage limits provide some stability: they create a "cushion" that can absorb some losses. The proof does not directly consider Eisenberg-Noe, but, using an argument analogous to [84, §5.2], it is possible to derive a sensitivity bound of $1/r$.

**Algorithm 7:** EGJ-COMMUNICATE-WITH(i)

```
1 begin
2     foreach j in neighbors(i) do
3         sendTo(j, 1-(value[i]/origVal[i]))
4     end
5 end
```

**Algorithm 8:** EGJ-AGGREGATE(i)

```
1 begin
2     totalShortfall = ∑_i ((value[i]<threshold[i]) ? (threshold[i]-value[i]) : 0)
3 end
```

### 3.4.5 Utility

This leaves two practical questions: 1) how frequently could these algorithms be safely executed, and 2) how does the addition of noise affect the utility of the output? We cannot hope to give final answers here because of the many policy decisions that would be involved, but we can at least provide ballpark figures.

First, we need to choose the privacy parameter $\varepsilon$. We assume that the banks would want to prevent an adversary from increasing their confidence in any fact about the input data by more than a factor of two; this yields $e^{\varepsilon_{\max}} = 2$ and thus a privacy "budget" of $\varepsilon_{\max} = \ln 2$.

Next, we need to calculate the amount of noise that would be added to the output. This depends on a) which input data sets would be considered similar (i.e., at what threshold $T$ the banks would wish to protect their financial data), and b) the sensitivity of the program. For a), we follow an argument from Flood et al. [66] and assume that a granularity of $T = \$1$ billion – roughly the size of the 100th largest bank's equity – is reasonable. For b), we use Elliot-Golub-Jackson as an example and set the leverage bound to $r = 0.1$, as mandated by the Basel III framework [19]. This yields a sensitivity of $2/r = 20$ (independent of the number of iterations); thus, the noise would be drawn from $T \cdot Lap(20/\varepsilon_{\text{query}})$.

Finally, we need to decide how precise the output needs to be, which controls the "privacy cost" $\varepsilon_{\text{query}}$ of the program. In 2015, the annual stress test mandated by Dodd-Frank yielded a TDS of about $500 billion [145], which was considered safe. We add a generous safety margin and assume that it would be sufficient to compute the TDS to within ±$200 billion. To ensure that the noise is lower than that with at least 95% confidence, we must choose $\varepsilon_{\text{query}} \geq 0.23$.

Since banks must retrospectively disclose their aggregate positions every year anyway, it seems reasonable to replenish the privacy budget once per year. Thus, it seems safe to execute Elliot-Golub-Jackson up to $(\ln 2)/0.23 \approx 3$ times per year, which is more frequent than today's annual stress tests.

### 3.4.6 Threat model

Recall from Section 3.3.2 that DStress assumes that the parties are honest but curious (HbC). At first glance, it is not obvious that this assumption holds universally in the financial world. However, recall that banks are heavily regulated, and that they already have to submit to audits today. It should be possible to use these audits to verify that the banks a) contribute accurate information, and that they b) correctly perform the steps of the DStress algorithm. For privacy reasons, today's audits are compartmentalized – that is, each auditor gets to see only the information from a single bank – but this is sufficient for our purposes: each auditor only needs to verify the steps that are taken by the specific bank she is responsible for,

## 3.5 Evaluation

In this section, we report results from our experimental evaluation of DStress. Our main goal is to determine whether DStress's costs are low enough for our application scenario, and whether it is sufficiently scalable.

### 3.5.1 Prototype and experimental setup

For our experiments, we built a prototype of DStress that consists of three components: 1) the Wysteria MPC runtime [143], which is based on an implementation of the GMW protocol [77] provided by Choi et al. [47]; 2) a distributed execution engine for graph algorithms; and 3) an implementation of the communication protocol from Section 3.3.5, based on the cryptographic primitives in the OpenSSL library. Our prototype generates Laplace noise using a circuit design from Dwork et al. [59]. To save computation and bandwidth, we apply a widely used ElGamal optimization [102] that reuses the same ephemeral key for each of the $L$ bits in the share but requires $L$ different public keys. Excluding Wysteria, our prototype consists of 11,904 lines of Java and 953 lines of C.

Unless otherwise noted, we conducted our experiments on Amazon EC2. We used up to 100 `m3.xlarge` instances, which each have four virtual Intel Xeon E5-2670 v2 2.5 GHz CPUs, 15 GB of memory, and two 40GB partitions of SSD-based storage. All the instances were located in the same EC2 region. For elliptic curves, we selected the NIST/SECG curve over a 384-bit prime field (`secp384r1`); this offers security equivalent to 192-bit symmetric cryptography, which is more than enough to defend against current cryptanalytic capabilities. We kept the default parameters for Wysteria and GMW: shares had a length of 12 bits (stored as 13 bytes), and the statistical security parameter for GMW was $k = 80$.

### 3.5.2 Microbenchmarks: Computation

DStress contains two main sources of computation cost: the MPC invocations that are used to perform the steps of the graph algorithm, and the cryptographic operations in the communication protocol. We evaluate each in turn.

**MPC invocations:** DStress performs four different kinds of operations in MPC: 1) the initialization step that generates the shares of each node's initial state; 2) the graph algorithm's computation step; 3) the graph algorithm's aggregation step; and 4) the

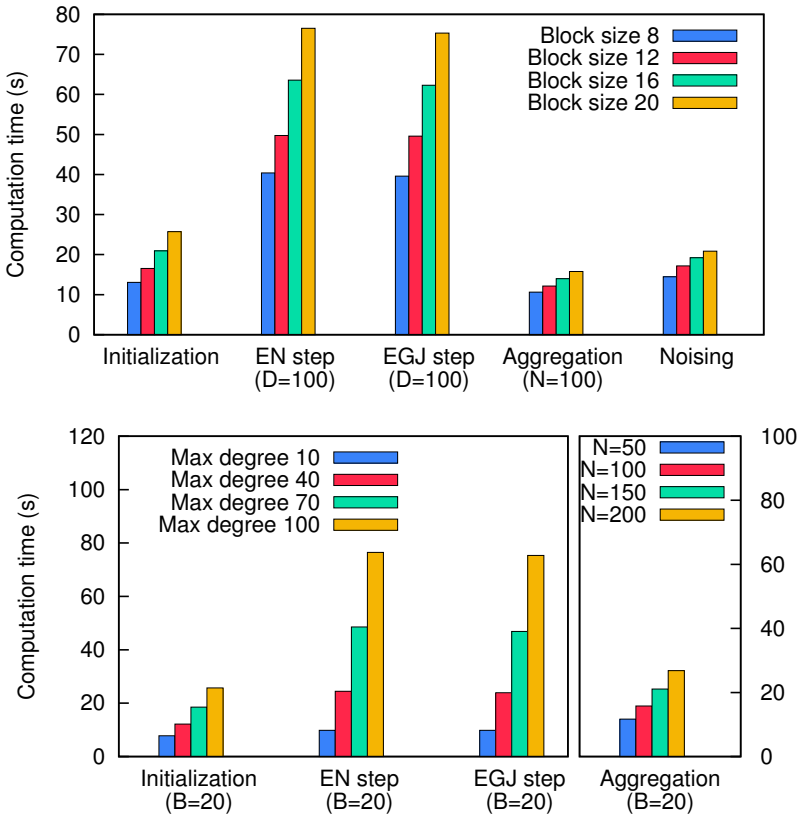



Figure 3.2: Computation time spent on MPC, with different block sizes (top), and different values for $D$ and $N$ (bottom).

final addition of Laplace noise. To quantify the cost of each, we performed a series of microbenchmarks in which we ran each MPC in isolation, using only Wysteria, for different block sizes. Since the computation steps in Eisenberg-Noe (EN) and Elliot-Golub-Jackson (EGJ) are different, we ran two separate experiments for this step.

The left part of Figure 3.2 shows the end-to-end completion times varied with the block size. There is a linear dependence, which is consistent with the theoretical complexity of GMW (the total cost is quadratic, but the nodes are working in parallel). We note that a block size of 20 is plausible in our setting: recall that the block size must be greater than the collusion bound $k$, and, to our knowledge, the largest known instance of collusion in the banking world was the LIBOR scandal, which involved 16 banks [144].

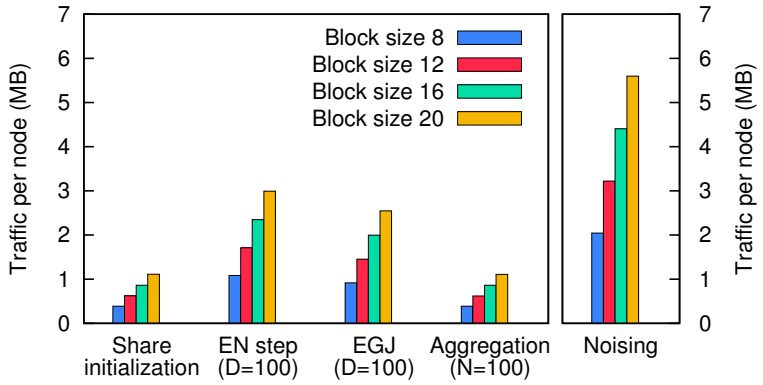



Figure 3.3: Per-node traffic generated by MPC computation steps with different block sizes.

The bottom part of Figure 3.2 shows how the time for the initialization and computation steps varied with the degree bound $D$, and how the time for the aggregation step varied with the number of nodes $N$. Again, the dependencies are roughly linear; this is because the corresponding MPC circuits are fairly simple, so the number of gates depends mostly on the number of inputs.

**Message transfers:** To quantify the cost of the message transfer protocol from Section 3.3.5, we measured the time needed to transfer a single 12-bit message between two blocks of different sizes. We found that the end-to-end completion time was roughly proportional to $k$, from 285 ms with an 8-node block to 610 ms with a 20-node block. This is expected because each node in the block must encrypt $k+1$ subshares. There is a quadratic component as well because a single node must combine the $(k+1)^2$ encrypted subshares using the additive homomorphism, but this involves simple multiplications; the cost is dominated by the exponentiations, which are far more expensive.

### 3.5.3 Microbenchmarks: Bandwidth

To quantify DStress's bandwidth cost, we measured the average amount of traffic that each node generated during the microbenchmarks from Section 3.5.2. As before, we examine the MPC and message transfer steps separately.

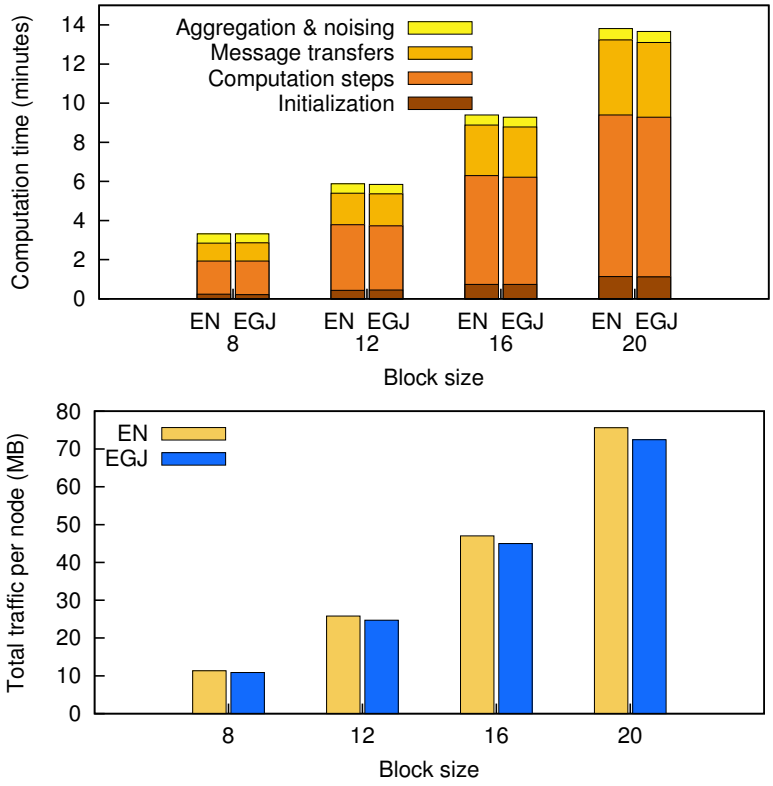



Figure 3.4: Computation time (top) and per-node traffic (bottom) for an end-to-end run on $N = 100$ vertexes (with maximum degree $D = 10$), while performing $I = 7$ iterations of Eisenberg-Noe (EN) and Elliot-Golub-Jackson (EGJ).

**MPC invocations:** Figure 3.3 shows our results for each of the five MPC circuits we have identified in Section 3.5.2. The traffic per node is roughly proportional to the block size $k+1$. Again, this is expected: although the total amount of traffic in GMW increases quadratically with the number of participants, the load is shared by $k+1$ nodes. We note that the absolute numbers are low and never exceed 6 MB per node, even for the comparatively large noising circuit. This is because Wysteria's GMW implementation includes oblivious transfer extensions [90, 108] as an optimization.

**Message transfers:** The amount of traffic for message transfers varies with the roles of the nodes. When the protocol is invoked for an edge $(i, j)$, node $i$'s load is the highest, since it receives $(k+1)^2$ encrypted subshares from $B_i$. In our experiments, this amounted to between 97 kB (with 8-node blocks) and 595 kB (with 20-node blocks). The nodes in $B_i$ each send $k+1$ encrypted subshares, and $j$ sends $k+1$

encrypted shares; thus, their traffic is linear in $k$ and never exceeded 29 kB per node in our experiments. The nodes in $B_j$ each receive a single encrypted share, regardless of the block size, so they handle a constant amount of traffic, about 1.4 kB.

Since we ran our experiments on EC2, neither propagation delays nor bandwidth constraints were major factors. This would be different in a wide-area deployment; however, since both MPC and the message transfers use relatively little bandwidth, we do not expect the network to become a major bottleneck in a wide-area setting.

### 3.5.4 End-to-end cost

To get a sense of the total cost of a DStress execution, we performed end-to-end runs with both EN and EGJ, using a synthetic graph with $N = 100$ banks, a degree limit of $D = 10$, and $I = 7$ iterations. As before, we varied the block size, and we measured the completion time and the average amount of traffic that was sent by each node.

Figure 3.4 shows our results. Although, as we have seen, the runtime of the individual operations is linear in $k$, the overall runtime varied roughly with $O(k^2)$; this is because, if we keep the number of nodes $N$ constant while increasing $k$, each node must also participate in more blocks. (The actual dependence is not perfectly quadratic because each node handles multiple blocks in parallel, and the corresponding computations can be overlapped when one of them blocks.)

### 3.5.5 Scalability

In 2015, there were roughly 1,750 large commercial banks in the United States [124]. Due to our limited budget, we were unable to perform experiments with that many nodes; instead, we estimate the cost using results from our microbenchmarks.

Given values for the degree bound $D$, the number of nodes $N$, the collusion bound $k$, and the number of iterations $I$, it is easy to estimate the cost of the initialization, computation, and communication steps. For aggregation, we assume a two-level aggregation tree with degree 100 – that is, DStress would first aggregate the values

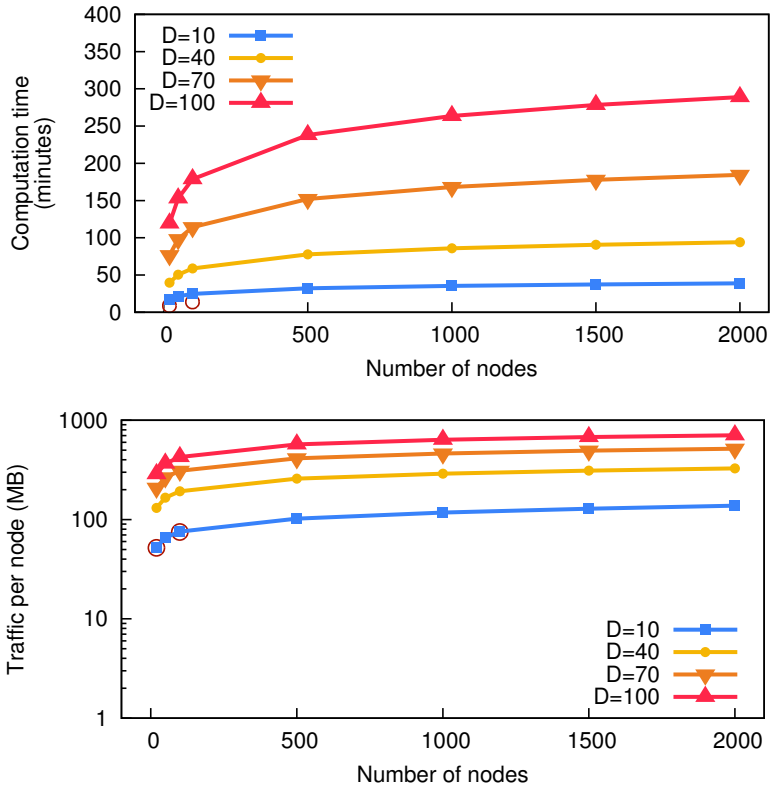



Figure 3.5: Projected computation cost (top) and per-node traffic (bottom) for end-to-end runs of EN on networks of different sizes. The red circles are validation points from two actual runs on EC2, with N=20 and N=100 nodes, respectively, and with D=10.

from groups of 100 nodes (in parallel) and then further aggregate the results before the final noising step. We conservatively use a degree bound of $D = 100$ and a block size of $k + 1 = 20$, and we assume that nodes cannot overlap computations from different blocks.

Obtaining realistic values for $I$ is nontrivial because the exact structure of the banking network is not known, and cannot be fully inferred from the public (aggregate) disclosures. However, work in theoretical economics [48] has shown how to infer at least an approximate graph from public data. We reconstructed graphs based on this work, and found that $I = \log_2 N$ is enough to allow the algorithm to converge. The relevant details are available in Appendix B.3.

Figure 3.5 shows our estimates for different network sizes; the red circles show

the results from actual EC2 runs ($N$=20 and $N$=100) we performed for validation. (Recall that actual runs tend to be a bit faster than predicted because of the overlap between different block computations.) Based on these results, we estimate that an end-to-end run of Eisenberg-Noe for the entire U.S. banking system ($N = 1,750$, $D = 100$) would take about 4.8 hours and consume about 750 MB of traffic. These costs seem low enough to be practical.

We did not compare DStress to prior work because we are not aware of any other system that efficiently offers guarantees that are comparable to those of DStress. However, one plausible baseline approach is to naïvely perform the entire computation as a single, monolithic MPC. If we ignore the details (such as the prorating, the comparisons, and the final matrix inversion), the closed form of an algorithm like EN essentially raises an $N \times N$ matrix to the $I$.th power. To estimate how long this would take, we wrote a simple Wysteria program that multiplies two square matrices, and we ran it for different values of $N$. As expected, the end-to-end completion time rose quickly, from 1.8 minutes for $N = 10$ to 40 minutes for $N = 25$. This is expected because the asymptotic complexity of matrix multiplication is $O(N^3)$. (Note that data-dependent optimizations cannot be applied because the data in the matrix is private.) We were unable to run the experiment for $N > 25$ because Wysteria ran out of memory, but we extrapolate that raising a 1750x1750 matrix to the $I - 1 = 11$th power would take $(1750/25)^3 * 40 * 11$ minutes, or about 287 years. This suggests that systemic risk detection using plain MPC would be infeasible in practice.

## 3.6 Related work

**Differential privacy:** There is a rich body of work on differentially private analytics for relational data [58], but there are much fewer results for graph data. [96] presents some private algorithms that offer *edge*-differential privacy, including $k-$triangle counting and $k-$star counting, but it is often difficult to give good accuracy with this approach because many algorithms have a high sensitivity to edge changes. Re-

stricted sensitivity [34] takes advantage of the queriers' prior beliefs about the data to achieve higher accuracy. Our systemic-risk case study uses a slightly different guarantee, *dollar*-differential privacy, which was first proposed by Flood et al. [66, §4.3].

**Distributed query processors:** The first differentially private query processors, such as PINQ [117], Fuzz [82], and Airavat [146] assumed a centralized setting in which the analyst has access to all the private data. Later systems added support for distributed data, but they typically focus on a specific class of queries: for instance, PDDP [45] can build histograms, and DJoin [121] can process certain types of joins. Narayan et al. [122] sketched a system that can run iterative graph algorithms but offers weaker privacy guarantees than DStress – for instance, it leaks the some information about the structure of the graph. To our knowledge, DStress is the first practical system to support iterative graph algorithms with strong differential privacy guarantees.

**Secure Multiparty Computation:** Most practical MPC implementations are either based on the GMW protocol [77], which expresses computations as boolean circuits [47, 29], or based on the BGW protocol [30], which expresses computations as arithmetic circuits [166, 43, 51]. A direct comparison between protocols of the two main strands is not straightforward. In general, systems that use BGW, such as PICCO [166] or SEPIA [43], can offer better performance for applications that mostly use arithmetic operations. However, not all applications are of this type: for instance, Choi et. al. [47] showed that boolean-circuit systems outperform arithmetic-circuit systems for a specific class of matching algorithms. The best appropriate choice of MPC protocol for systemic risk algorithms is an open question; we selected GMW and [47] because both EGJ and EN can be expressed as graph computations, which seem to be a closer match to the algorithms described in [47]. In principle, our approach – breaking up a large MPC computation into smaller computations – should be applicable to the BGW protocol as well.

Recently, work on two-party secure computation (2PC) has started considering graph computations as well [127, 126]. Nayak et al. [126] identifies two key challenges in extending secure computation to graphs: one needs to protect the privacy of data as well as the graph topology. The solution described in [126] achieves the goals by obliviously sorting a combined list of all graph vertices and edges using garbled circuits. Unfortunately, full MPC is several orders of magnitude slower than 2PC; hence, this approach would face the same efficiency challenges that we detailed in 3.2.2.

We emphasize that we are not the first to consider the use of MPC for *differentially* private computations (see, e.g., [59, 25]). Our contribution is an efficient, scalable protocol for executing graph algorithms in a distributed setting without revealing the structure of the graph.

**Message transfer protocol:** Using ElGamal for its key randomization property has been considered in the literature before [80, 72, 56]. In fact, work concurrent to ours [56] presents an ElGamal-based construction which is similar to our message transfer protocol. Unfortunately, that solution is not additively homomorphic and cannot be directly used in DStress.

**Distributed ledgers for financial networks:** Corda [7, 119] is a distributed ledger system designed to record transactions between financial institutions. The recorded transactions are cryptographically verified, so the ledger could be used to provide evidence and aid the resolution of legal disputes between banks. Since the validity of transactions depends on other transactions, verification involves multiple parties, and information about sensitive transactions can leak [119, §4.2]. Even though Corda prevents nodes from seeing transactions that do not require their verification signature, it does not provide strong privacy guarantees about what these nodes learn [119, §4.2, §15].

## 3.7 Conclusion

In this Chapter, we have presented DStress, a system that can efficiently analyze large, distributed graphs with confidential information. DStress's programming model resembles that of other frameworks for graph analytics; however, DStress executes programs in a distributed fashion, using a combination of secret sharing, small multi-party computations, and a special protocol for transferring messages without revealing the structure of the graph. As a result, DStress only needs a few hours to run computations with hundreds of participants, whereas a naïve application of multi-party computation would take many years.

We have also studied one concrete use case of DStress that we have taken from the economics literature: the computation of systemic risk in financial networks. We have shown that DStress can implement two state-of-the-art models of systemic risk; our experimental results suggest that these models could be evaluated on all the large commercial banks in the United States within about five hours, using only one commodity machine at each bank.

# 4

# Queries under the trusted hardware assumption

## 4.1 Introduction

Recently, a number of systems have been proposed that can provide *privacy-preserving distributed analytics* [147, 168]. At a high level, these systems provide functionality that is comparable to a system like Spark [164]: users can upload large data sets, which are distributed across a potentially large number of nodes, and they can then submit queries over this data, which the system answers using a distributed query plan. However, in contrast to Spark, these systems *also* protect the confidentiality of the data. This is attractive, e.g., for cloud computing, where the owner of the data may wish to protect it against a potentially curious or compromised cloud platform.

It is possible to implement privacy-preserving analytics using cryptographic techniques [138, 134], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach – which was recently applied in Opaque [168] – is to rely on *trusted hardware*, such as Intel's SGX. With this approach, the data remains encrypted even in memory and is only

accessible within a trusted *enclave* within the CPU. As long as the CPU itself is not compromised, this approach can offer very strong protections, even if the adversary has managed to compromise the operating system on the machines that hold the data.

However, even though SGX-style hardware can prevent an adversary from observing the data itself, the adversary can still hope to learn facts *about* the data by monitoring various side channels. The classic example is a timing channel [103]: suppose a query computes the frequency of various medical diagnoses, and the adversary knows that the computation will take $51\mu$s if Bob has cancer, and $49\mu$s otherwise. Then, merely by observing the amount of time that is spent in the enclave, the adversary can learn whether or not Bob has cancer. Other common side channels that have been exploited in prior work include the sequence of memory accesses from the enclave [162], the number and size of the messages that are exchanged between the nodes [129], the contents of the cache [42], and the fact that a thread exits the enclave at a certain location in the code [106, 161].

Today, system designers have basically two options for dealing with side channels, and neither of them is particularly attractive. The first option is to simply exclude some or all of these channels from the threat model: for instance, Opaque [168] explicitly declares timing channels to be out of scope. This is not very satisfying: while timing channels intuitively "do not leak very much", prior work shows that they can in fact leak quite a bit, such as entire cryptographic keys [167]. The second option is to plug the channels by enforcing complete determinism, e.g., by using oblivious algorithms [13] and by padding computation time and message size all the way to their worst-case values. This approach is safe but even less satisfying: as we will show experimentally, full padding can drive up the overhead by several orders of magnitude.

In this Chapter, we propose a more principled approach, which consists of three parts. The first is a primitive that can perform small computations safely, by exe-

cuting them in a core that is completely "locked down" and cannot be interrupted or access uncached data during the computation. This primitive, which we call an *oblivious execution environment (OEE)*, protects against most realistic side channels and yet improves efficiency, since there is no need to use oblivious algorithms while the core is locked. The second element is a set of enhanced oblivious operators that plug not only the memory access channel – like traditional oblivious operators – but also plug or limit the three other side channels we discussed above. The third element is a query planner that combines several of these smaller computations to answer larger queries. To avoid high overheads, our query planner is allowed to release *some* information about the private data, but only in a carefully controlled fashion, using differential privacy [60]. By combining these three elements, it is possible to answer queries efficiently, while at the same time giving strong privacy guarantees.

We have implemented our approach in a system we call Hermetic. Since the current SGX hardware is not yet able to fully support the "lockdown" primitive we propose, we have implemented the necessary functionality in a small hypervisor that can be run on commodity machines today. Our results from a detailed experimental evaluation of Hermetic show that our approach is indeed several orders of magnitude more efficient than full padding (which is currently the only approach that can reliably prevent side channels). The overheads are comparable to those of existing SGX-based distributed analytics systems.

We note that Hermetic is not a panacea: like all systems that are based on trusted hardware, it assumes that the root of trust (in the case of SGX, Intel) is not compromised. Also, there are physical side channels that even Hermetic cannot plug: for instance, an adversary could use power analysis [99] or electromagnetic emanations [101], or simply depackage the CPU and attack it with physical probes [152]. However, these attacks are much harder to carry out, require specialized equipment, and may be impossible to eliminate without extensive hardware changes. Our con-

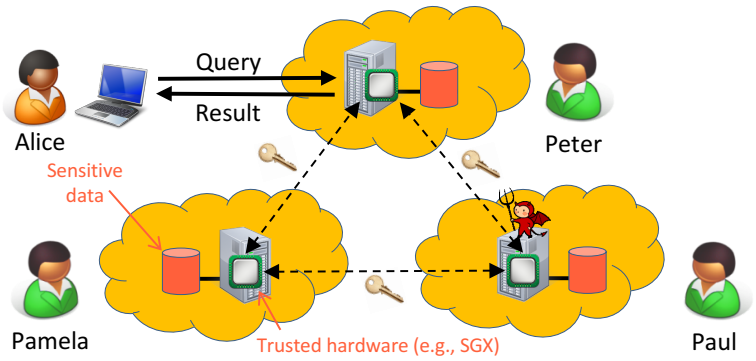



Figure 4.1: Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU.

tributions are as follows:

- The OEE primitive, which performs simple computations privately, using a locked-down core (Section 4.3);
- A new set of oblivious operators that prevent or limit four different side channels (Section 4.4);
- A novel privacy-aware query planner (Section 4.5);
- The design of the Hermetic system (Section 4.6);
- A prototype implementation of Hermetic (Section 4.7); and
- A detailed experimental evaluation (Section 4.8)

## 4.2 Overview

Figure 4.1 illustrates the scenario we are interested in. There is a group of *participants*, who each own a sensitive data set, as well as a set of *nodes* on which the sensitive data is stored. An *analyst* can submit queries that can potentially involve data from multiple nodes. Our goal is to build a distributed database that can answer these queries *efficiently* while giving strong privacy guarantees to each participant. We assume that the queries themselves are not sensitive – only their answers are – and that each node contains a trusted execution environment (TEE) that supports secure

enclaves and attestation, e.g., Intel's SGX.

Note that this scenario is a generalization of the scenario in some of the earlier work [147, 168], which assumes that there is only one participant, who outsources a data set to a set of nodes, e.g., in the cloud.

### 4.2.1 Threat model

We assume that some of the nodes are controlled by an adversary – for instance, a malicious participant or a third party who has compromised the nodes. The adversary has full physical access to the nodes under her control; she can run arbitrary software, make arbitrary modifications to the OS, and read or modify any data that is stored on these nodes, including the local part of the sensitive data that is being queried. We explicitly acknowledge that the analyst herself could be the adversary, so even the queries could be maliciously crafted to extract sensitive data from a participant.

### 4.2.2 Background: Differential privacy

One way to provide strong privacy in this setting is to use *differential privacy* [60], one of the strongest known privacy guarantees. Differential privacy is a property of *randomized* queries; that is, queries do not compute a single value but rather a probability distribution over the range $R$ of possible outputs, and the actual output is then drawn from that distribution. This can be thought of as adding a small amount of random "noise" to the output. Intuitively, a query is differentially private if a small change to the input only has a statistically negligible effect on the output distribution.

More formally, let $I$ be the set of possible input data sets. We say that two data sets $d_1, d_2 \in I$ are similar if they differ in at most one element. A randomized query $q$ with range $R$ is $\varepsilon$-differentially private if, for all possible sets of outputs $S \subseteq R$ and

all similar input data sets $d_1$ and $d_2$,

$$Pr[q(d_1) \in S] \leq e^{\varepsilon} \cdot Pr[q(d_2) \in S].$$

That is, any change to an individual element of the input data can cause at most a small multiplicative difference in the probability of *any* set of outcomes $S$. The parameter $\varepsilon$ controls the strength of the privacy guarantee; smaller values result in better privacy. For more information on how to choose $\varepsilon$, see, e.g., [85].

Differential privacy has strong composition theorems; in particular, if two queries $q_1$ and $q_2$ are $\varepsilon_1$- and $\varepsilon_2$-differentially private, respectively, then the combination $q_1 \cdot q_2$ is $\varepsilon_1 + \varepsilon_2$-differentially private. Because of this, it is possible to associate each data set with a "privacy budget" $\varepsilon_{\max}$ that represents the desired strength of the overall privacy guarantee, and to then keep answering queries $q_1, \ldots, q_k$ as long as $\sum_i \varepsilon_i \leq \varepsilon_{\max}$. (Note that it does not matter what specifically the queries are asking.)

### 4.2.3 Strawman solution with TEEs

At this point, it may appear that the problem we motivated above could be solved roughly as follows: each node locally creates a secure enclave that contains the database runtime, and the participants use attestation to verify that the enclaves really do contain the correct code. Each participant $P_i$ then opens a secure connection to her enclave(s) and uploads her data $d_i$, which is stored in encrypted form, and then sets a local privacy budget $\varepsilon_{\max,i}$ for this data. When the analyst wishes to ask a query, he must create and submit a distributed *query plan*, along with a proof that the query is $\varepsilon_i$-differentially private in data set $d_i$; the enclaves then verify whether a) they all see the same query plan, and b) there is enough privacy budget left – that is, $\varepsilon_{\max,i} \geq \varepsilon_i$ for each $d_i$. If both checks succeed, the enclaves execute the query plan, exchanging data via encrypted messages when necessary, and eventually add the requisite amount of noise to the final result, which they then return to the analyst.

This approach would seem to meet our requirements: differential privacy ensures that a malicious analyst cannot compromise privacy, and the enclaves ensure that compromised nodes cannot get access to data from other nodes or to intermediate results.

### 4.2.4 Problem: Side channels

However, the strawman solution implicitly assumes that the adversary can learn *nothing at all* from the encrypted data or from externally observing the execution in the enclave. In practice, there are several *side channels* that remain observable; the most easily exploitable are:

- **Timing channel (TC) [103]**: The adversary can measure how long the computation in the enclave takes, e.g., by reading a cycle-level timestamp counter in the CPU before entry and after exit;
- **Memory channel (MC):** The adversary can observe the locations in memory that the enclave reads or writes (even though the data itself is encrypted!), as well as their timing, e.g., by inspecting the page tables or by measuring the contents of the cache; and
- **Instruction channel (IC):** The adversary can see the sequence of instructions that are being executed, e.g., by looking at instruction cache misses; and
- **Object size channel (OC):** The adversary can see the size of any intermediate results that the enclave stores or exchanges with other enclaves.

At first glance, these channels may not reveal much information, but this intuition is wrong: prior work has shown that side channels can be wide enough to leak entire cryptographic keys within a relatively short amount of time [167]. To get truly robust privacy guarantees, it is necessary to close or at least mitigate these channels.

### 4.2.5 State of the art

Prior work has generally approached side channels in one of three ways. The first is to simply exclude side channels from the threat model, as, e.g., in [168]. This seems fine if the work primarily focuses on some other challenge, but the effective privacy guarantees remain weak until the solution is combined with some defense against side channels (such as the one we propose here). The second is to use heuristics to reduce the bandwidth of some of the channels, as, e.g., in [16, 86, 95]. This approach is stronger than the first, but it remains somewhat unsatisfying, since it is difficult to formally reason about the true strength of the guarantee.

The third, and most principled, approach is to remove the dependency between the sensitive data and the signal that the adversary can observe. For instance, oblivious algorithms [13] can close the memory channel by accessing the data in a way that depends only on the size of the data, but not on any specific values, and full padding can close the timing channel by padding the computation time to its worst-case value. These approaches work well, but they have two important drawbacks: 1) to our knowledge, all prior solutions close only some subset of our four channels, but not all four; and 2), as we will show experimentally in Section 4.8, these approaches can have an enormous overhead, sometimes by several orders of magnitude. We are not aware of an efficient, principled solution that can close all four of our channels simultaneously.

### 4.2.6 Approach

Our goal in this Chapter is to get the "best of both worlds": we want to reliably close the four side channels from Section 4.2.4 while preserving reasonably better performance. Our approach is based on the following three ideas:

**Oblivious execution environments:** We provide a primitive called OEE that can perform small computations $out := f(in)$ entirely in the CPU cache, and such that both the execution time and the instruction trace depend only on $|in|$, $|out|$, and $f$,

but not on the actual data. This completely avoids all four channels.

**New oblivious operators:** We present several oblivious operators that can be used to compose OEE invocations into larger query plans. In addition to avoiding the memory channel – like all oblivious algorithms – our operators also have a deterministic control flow (to avoid the instruction channel), they use only timing-stable instructions (to avoid the timing channel), and the size of their output is either constant or noised with "dummy rows" to ensure differential privacy (to avoid the object size channel). Notice that there is an efficiency-privacy tradeoff: more noise yields better privacy but costs performance.

**Privacy-aware query planner:** We describe a query planner that optimizes for both efficiency *and* privacy, by carefully choosing the amount of noise that is added by each oblivious operator.

## 4.3 Oblivious Execution Environments

The first part of Hermetic's strategy to mitigate side channels is hardware-assisted oblivious execution, using a primitive we call an *oblivious execution environment (OEE)*.

### 4.3.1 What is oblivious execution?

The goal of oblivious execution is to compute a function $out := f(in)$ while preventing an adversary from learning anything other than $f$ and the sizes $|in|$ of the input and $|out|$ of the output - *even if*, as we have assumed, the adversary can observe the memory bus and the instruction trace, and can precisely measure the execution time.

Some solutions for oblivious execution already exist. For instance, Arasu et al. [13] use special oblivious sorting algorithms that perform a fixed set of comparisons (regardless of their inputs) and thus perform memory accesses in a completely deterministic way; also, compilers have been developed that emit code without data dependent branches [142], and thus have a perfectly deterministic instruction trace.

However, the existing approaches have three key limitations. First, they are very inefficient – largely because they must perform numerous dummy memory accesses to disguise the real ones or employ expensive oblivious RAMs [76, 155, 154]. Second, they tend to focus on one or two specific side channels; for instance, even if the memory trace is deterministic, the execution time can still vary if the code includes instructions whose execution time is data-dependent. Third, the existing solutions all implicitly assume that the adversary cannot interrupt the execution and inspect the register state of the CPU.

### 4.3.2 Oblivious execution environments

To provide a solid foundation for oblivious execution, we introduce a primitive OEE (f,in,out) that, for a small set of predefined functions $f$, has the following three properties:

1. Once invoked, OEE runs to completion and cannot be interrupted or interfered with;
2. OEE loads *in* and *out* into the cache when it starts, and writes *out* back to memory when it terminates, but does not access main memory in between;
3. The execution time, and the sequence of instructions executed, depend only on $f$, $|in|$, and $|out|$; and
4. The final state of the CPU depends only on $f$.

A perfect implementation of this primitive would plug all four side channels in our threat model: The execution time, the sequence of instructions, and the sizes of the *in* and *out* buffers are constants, so no information can leak via the TC, IC, or OC. Also, the only memory accesses that are visible on the memory bus are the initial and final loads and stores, which access the entire buffers, so no information can leak via the MC. Finally, since the adversary cannot interrupt the algorithm, she can only observe the final state of the CPU upon termination, and that does not depend on the data.

Note, however, that OEE is allowed to perform data-dependent memory accesses *during* its execution. Effectively, OEE is allowed to use a portion of the CPU cache as a private, un-observable memory for the exclusive use of $f$. This is what enables Hermetic to provide good performance.

### 4.3.3 Challenges in building an OEE today

Some of the properties of an OEE can be achieved through static transformations: for instance, we can (and, in our implementation, do) achieve property #3 by eliminating data-dependent branches and by padding the execution time to an upper bound via busy waiting. We can also disable hardware features such as hyperthreading that would allow other programs to share the same core, and thus potentially glean some timing information from the OEE. Properties #2 and #4 can be achieved through careful implementation. Finally, by executing the OEE in a SGX enclave, we can ensure that the data is always encrypted while in memory.

However, today's SGX unfortunately *cannot* be used to achieve property #1. By design, SGX allows the OS to interrupt an enclave's execution at any time, as well as flush its data from the cache and remove its page table mappings [49]. Indeed, these limitations have already been exploited to learn the secret data inside enclaves [162, 42, 106].

### 4.3.4 The Hermetic hypervisor

To overcome these limitations, we use a small hypervisor. Before an OEE can execute, the hypervisor (1) completely "locks down" the OEE's core by disabling all forms of preemption – including IPIs, IRQs, NMIs, and timers; (2) prevents the OS from observing the OEE's internal state, by mapping or flushing any of its memory pages, or by accessing hardware features such as performance monitoring or hardware breakpoints; and (3) returns control to the enclave, so it can prefetch *in* and perform dummy writes to *out* and to the stack, so that both are in the cache and the

cache lines of the latter are already in the Modified state. When the OEE completes, the cache is flushed, and the hypervisor re-enables preemption.

If the OEE's core shares a last-level cache with other cores, the hypervisor must take care to prevent cache timing attacks. One way to do this would be to simply lock down these other cores as well; however, this would severely limit concurrency while an OEE is executing. Instead, we can use Intel's Cache Allocation Technology (CAT) [97] to partition the cache between the OEE's core and the other cores at the hardware level. In Section 4.7, we present further details of the hypervisor's design and its use of the CAT.

We view the hypervisor as interim step that makes deploying Hermetic possible today. Its functionality is constrained enough that it could be subsumed into future versions of SGX. We believe that this Chapter and other recent work on the impact of side channels in TEEs demonstrates the importance of adding OEE functionality to TEEs.

## 4.4 Oblivious operators

OEEs provide a way to safely execute simple computations on small amounts of data, without side channels. However, to answer complex queries over larger data, Hermetic also needs higher-level operators, which we describe next.

### 4.4.1 Background

Prior work [13, 130] has already developed a number of oblivious operators, which are guaranteed to access the data in a deterministic way. A simple example would be a projection operator that performs a linear scan over all the input tuples and extracts a particular column of interest. This already addresses a subset of the MC, since the data is accessed in a deterministic, data-independent order, so we use these operators as a starting point. However, note that there are other kinds of memory accesses (to code, to the stack, etc.), and that the *timing* of the accesses can be disclosive as well;

we will discuss how we address these shortly.

Previous work includes some operators that are relatively standard; for instance, `project`, `rename`, `union`, and `cartesian-product` are similar to the operators found in any standard DBMS, and they are oblivious by nature, e.g., `cartesian-product` considers all different pairs of tuples from two relations in a data-independent way. However, there are some additional primitives that are needed for oblivious operation. The linchpin of oblivious query processing is an oblivious `sort` primitive. Classical sorting algorithms, such as Quicksort, would leak the ordering of the data elements, so this requires special algorithms, such as Batcher's odd-even mergesort (`batcher-sort`) [22], that access the data in a deterministic order.

Building on oblivious sorting, one can implement a variety of low-level oblivious primitives. `augment` adds a new column to a relation and sets it to the value of a particular expression; `grsum` (group-running sum) adds up the values in a column for fields that share the same key; `filter` discards all tuples that do not satisfy a predicate; `semijoin-aggregation` counts the occurrences of one relation's attribute in another relation; `expand` creates $k$ clones of each tuple, where $k$ is a value that is taken from a special column; and `stitch` linearly "stitches together" two relations of the same size by combining their columns.

Higher-level operators can be implemented by combining the above primitives. For instance, to implement a `join`, one first uses `semijoin-aggregation` on each relation to compute how many matches each tuple has in the other relation, then performs an `expand` to create as many clones of each tuple, and applies `stitch` to create the result. Similarly, a `groupby` would first `sort` the relation by the group key, then apply `grsum` to add up the values in the column that is being aggregated, and finally use `filter` to leave only the last tuple for each key, which contains the total sum. These primitives, as well as the algorithms for building larger query plans from these smaller primitives, are fairly standard, so we do not discuss them in detail; instead, we focus on the points where Hermetic differs from prior work.

### 4.4.2 Challenges

There are three key reasons why the existing operators are not sufficient for our purposes. The first and simplest one is that prior work tends to specify the operators in pseudocode, whereas the TC and IC very much depend on the finer details of the implementation. For instance, some `x86` instructions, such as floating-point instructions and integer division, have data-dependent timing, and must be avoided to prevent the TC; similarly, to prevent the IC, we must avoid data-dependent branches (e.g., by using the `cmov` instruction) and preload the code and the stack. The necessary steps are known, and we do not claim them as a contribution.

The second reason is that some prior work [13] often assumes a client-server model: the data is stored on the server and queries are processed on the client, which can issue read and write requests to the server. The threat model typically limits the adversary to observing the *sequence* of reads and writes, which excludes the TC and IC entirely and limits the MC to only data accesses. In other words, accesses to code or the stack are not considered, and it is assumed that the adversary cannot observe the precise timing of the accesses. In Hermetic, we make use of the OEE to block these channels; however, since the OEE is limited to small data sets, we also need operators to handle larger workloads.

The third reason is that, to our knowledge, prior work pays almost no attention to the OC: a `select` either reveals the *exact* number of rows that match the predicate, which is disclosive, or massively pads the output to the worst-case size, which is inefficient. To address this, we introduce a new technique, which we discuss next.

### 4.4.3 Dummy rows

To address the OC, Hermetic can pad relations with *dummy tuples*, so they appear larger (to the adversary) than the actual data tuples they contain. In essence, we add a special `isDummy` column to each relation, which is set to 1 for dummy tuples and to 0 otherwise, and we augment all the oblivious operators to ignore tuples that have

`isDummy` set to 1. The latter is necessary to maintain correctness: for instance, we must prevent the dummy tuples from appearing in the output of a `select`, and we achieve this by adding an `&& !isDummy` to each predicate in the `filter`.

Hermetic also needs a way to introduce dummy tuples in all the operators that have a variable-size output (i.e., whose output size is not a deterministic function of their input size) – specifically, `select`, `groupby`, and `join`. We modified the above three operators to take an extra parameter that specifies the number of dummy tuples to add. For details please refer to Appendix C.2.

### 4.4.4 Differential privacy

To ensure differential privacy, the number of dummy tuples that a given operator must add needs to be drawn from a Laplace distribution with parameter $\lambda = s/\varepsilon$, where $\varepsilon$ is the privacy parameter (Section 4.2.2) and $s$ is the *sensitivity* of the output size to the input data – the maximum change in the output size that can result from adding or removing one input tuple. The sensitivity can be determined by the query planner (Section 4.5), but the actual Laplace value has to be drawn in the enclave, when the query is being executed. It is critical that this draw be immune to our four side channels as well: if the adversary can learn the value that is being drawn, she can compute the actual number of data tuples in the output and thus potentially learn facts about the input data.

To guard against this, we developed a special operator that can draw values from a Laplace distribution in constant time, with a deterministic control flow and without accessing main memory. To some degree, this involves only careful programming, e.g., to ensure that only static loop bounds are used; however, one important challenge is that the floating-point instructions on `x86` cannot be used because they have data-dependent timing [12]. We instead used a fixed-point math library based on [12], which uses integer instructions.

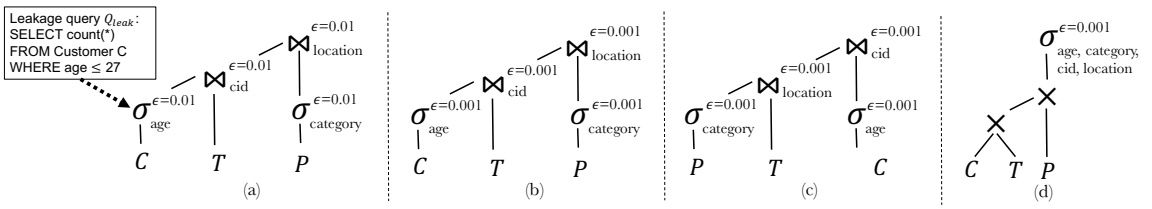



Figure 4.2: Different plans for query "`SELECT count(*) FROM C, T, P WHERE C.cid=T.cid AND T.location = P.location AND C.age≤27 AND P.category='hospital'`".

### 4.4.5 New primitives

Finally, Hermetic needs three additional oblivious primitives. The first two are `histogram`, which computes a histogram over the values in a given column, and `multiplicity`, which computes the multiplicity of a column – that is, the number of times that the most common value appears. These operators are used by the query planner to compute statistics about the input data that it needs to find the best privacy-efficiency tradeoff. We discuss these more in Section 4.5.

The third new primitive is `hybrid-sort`, which can obliviously sort large relations by repeatedly invoking OEE to sort smaller chunks of data using a fast, non-oblivious sorting algorithm (mergesort), and to combine the results by merging chunks of data using a non-oblivious merging algorithm (linear-merge). `hybrid-sort` is essentially a block-based variant of `batcher-sort` that takes advantage of the OEE, so the obliviousness properties of `batcher-sort` trivially apply to `hybrid-sort` as well.

## 4.5 Privacy-aware Query Planning

Next, we describe how Hermetic assembles the operators from Section 4.4 into query plans that can compute the answer to SQL-style queries. Query planning is a well-studied problem in databases, but Hermetic's use of differential privacy adds an interesting twist: Hermetic is free to choose the amount of privacy budget $\varepsilon$ it spends on each noised operation. Thus, it is able to make a tradeoff between privacy and performance: smaller values of $\varepsilon$ result in stronger privacy guarantees but also add

more dummy tuples, which slows down the downstream operators.

### 4.5.1 Computing operator sensitivities

For any query plan it considers, Hermetic must first derive upper bounds on the sensitivities of all the operators $o_i$ in the plan. To do this, Hermetic derives subqueries that compute the number of tuples in each operator's output; we call these query the *leakage queries* of the operators. Figure 4.2 illustrates how this is done for a simple example query (shown in the caption): the number of tuples that are output by the selection operator in Figure 4.2(a) is simply the number of customers who are at most 27 years old.

Once an operator's leakage query is known, Hermetic applies an algorithm from [121] to compute an upper bound on its sensitivity $s_i$. If the leakage query contains joins, the algorithm needs to know the multiplicities of the joined attributes; Hermetic obtains these using the `multiplicity` operator from Section 4.4.5. Once $s_i$ is known, Hermetic annotates the operator accordingly. If each operator adds a number of dummy tuples that is drawn from $Lap(s_i/\varepsilon_i)$, the overall query plan is $(\sum_i \varepsilon_i)$-differentially private.

However, drawing from $Lap(s_i/\varepsilon_i)$ can return negative values, but the padding must be positive to avoid deleting useful results. Hence, Hermetic actually adds $o_i + Lap(s_i/\varepsilon_i)$ dummy tuples, where $o_i$ is a tunable parameter. If the value drawn from $Lap(s_i/\varepsilon_i)$ at runtime is smaller than $-o_i$, the query fails and has to be retried. Notice that this means that Hermetic technically provides $(\varepsilon, \delta)$-differential privacy [59], a standard generalization of differential privacy.

### 4.5.2 Cost estimation

As discussed earlier, choosing the $\varepsilon_i$ values in a query plan involves a tradeoff between privacy and performance. The privacy cost of a query plan is simply $\sum_i \varepsilon_i$, so we focus on estimating the performance. To obtain a performance model, we derived

| Operator | Cost |
|---|---|
| HybridSort | $h(n) = n \cdot \log(c) + n \cdot \log^2(n/c)$ |
| Select | $h(n)$ |
| Group-by | $3 \cdot h(n)$ |
| Join | $4 \cdot h(n+m) + 2 \cdot h(m) + 3 \cdot h(n) + 2 \cdot h(k)$ |

Table 4.1: Performance model for the main relational operators in Hermetic.

the complexity of the Hermetic operators as a function of their input size; the key results are shown in Table 4.1. To estimate the size of intermediate results, we used an established histogram-based approach from the database literature [137]. According to this approach, the output size of selections is $N_R \cdot sel(R.a = X)$ and of joins is $N_{R_1} \cdot N_{R_2} \cdot sel(R_1.a \bowtie R_2.b)$, where $N_{R_i}$ is the size of input relation $R_i$, and $sel(R.a = X)$ and $sel(R_1.a \bowtie R_2.b)$ correspond to the estimated selectivities of selection and join, respectively. As shown in [83], the selectivities can be estimated from simple statistics that Hermetic computes using the `histogram` operator from Section 4.4.5.

To enable the planner to assess the performance implications of the dummy tuples, Hermetic takes them into account when estimating relation sizes. Since $Lap(s_i/\varepsilon_i)$ has a mean of zero, the expected number of dummy tuples added by operator $O_i$ is simply the offset $o_i$.

### 4.5.3 Query optimization

Hermetic's query planner uses relational algebra rules to generate all possible plans, and then picks an optimal plan based on the estimated privacy and performance costs. Since, in the applications we consider, privacy is usually more important than performance, we designed our optimizer to select the fastest plan whose privacy cost does not exceed a budget $\varepsilon_B$, which can be set by the analyst.

Figure 4.2 illustrates some of the decisions the query planner makes. The first decision has to do with choosing the $\varepsilon_i$ parameter of each operator, which affects both performance and privacy cost. Plans (a) and (b) show two plans that have the same structure but use a different $\varepsilon$ parameters; plan (a) achieves better performance

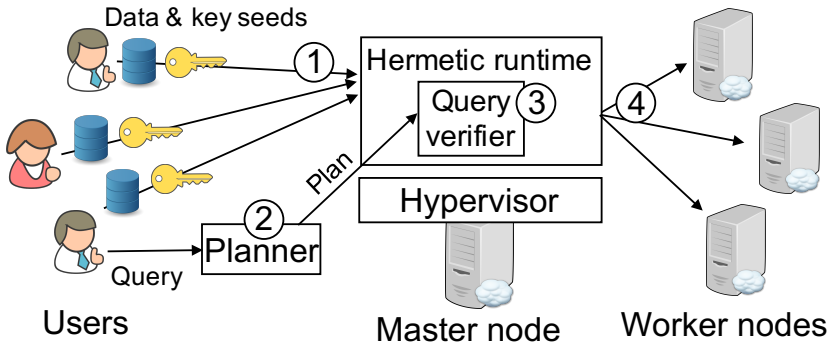



Figure 4.3: Hermetic's workflow.

because the larger $\varepsilon_i$ implies smaller intermediate results, whereas plan (b) consumes less privacy budget because the $\sum_i \varepsilon_i$ is smaller.

Another decision of the planner is illustrated by plans (b) and (c). These plans use the same $\varepsilon$, but the order of joins is different, and this affects how much $\varepsilon$ is consumed from the budget of each relation (e.g., plan (b) uses 0.003 from C's budget, but (c) uses 0.002). Finally, plan (d) uses Cartesian products, which do not consume any $\varepsilon$; thus, the plan achieves the smallest total $\varepsilon$ consumption.

Notice that $\varepsilon_i$ can be chosen independently for each operator, which gives rise to an interesting optimization problem. Our current implementation just tries a few standard values, but a more sophisticated optimizer could potentially find plans that are more private and/or more efficient.

## 4.6 The Hermetic System

Next, we describe Hermetic's design and how it combines the techniques described in the previous sections to execute relational queries on sensitive data.

### 4.6.1 Overview

Hermetic consists of a master node, and several worker nodes. Each node runs the trusted hypervisor (Section 4.3), and the trusted runtime that performs the Hermetic operators (Section 4.4) inside a secure enclave. The last component of Hermetic is

the query planner from Section 4.5. Figure 4.3 shows the workflow of Hermetic, which consists of the following steps:

1. Initially, the master node launches the Hermetic hypervisor and runtime, and the users contact the runtime to setup a master encryption key and upload their data (Section 4.6.2). Users are convinced of the authenticity of both the hypervisor and the runtime by using attestation (Section 4.6.3).
2. After initialization, users can submit queries to the Hermetic optimizer, which generates a concrete query plan and sends it to the runtime for execution (Section 4.6.4).
3. Since the optimizer is outside Hermetic's trusted code base, the runtime verifies incoming plans to make sure that all operators are annotated with the appropriate sensitivity and epsilon (Section 4.6.4).
4. The runtime asks Hermetic worker nodes to execute the operators of the query plan using Hermetic's oblivious operators. Afterwards, the results are returned to the user who submitted the query (Section 4.6.5).

Next, we describe these steps in greater detail.

### 4.6.2 Initialization

Hermetic is initialized after the users setup a master encryption key and upload their sensitive data to the server. Since no party in Hermetic is completely trusted, the master key is created inside the trusted runtime using randomness contributed by the users. After that, the key is encrypted using a hardware-based key and persisted to secondary storage using, e.g., Intel SGX's sealing infrastructure [91].

With the master key in place, users send their data to the runtime, which encrypts it with the key and stores it to the disk. Since the size of the sensitive data can reveal sensitive information too, we assume that users will pad the initial relations with noise before uploading them to the runtime. At the same time, the privacy budget of each uploaded relation is initialized and stored to the disk.

To protect from replay attacks [24, 114, 41], where an old version of the privacy budget is reused by the system, Hermetic uses a trusted non-volatile hardware counter to add a monotonic value to the privacy budget, which is then checked to confirm the budget is fresh. Such a counter is available in trusted hardware solutions, including the current version of SGX [89]. To enable the query optimizer to calculate the privacy and performance cost of query plans, some basic statistics are also computed during initialization.

### 4.6.3 Attestation

A prerequisite for uploading sensitive data is that users can be convinced that they are sending the data to a correct instantiation of the Hermetic system. This means that they need to make sure that the Hermetic hypervisor is running on the remote machine, and that the trusted hardware runs the Hermetic runtime. We achieve this level of trust as follows. Upon launch, the Hermetic runtime uses Intel's trusted execution technology (TXT) [92] to get an attestation of the boot process and the hypervisor loaded at the time. If the Hermetic runtime is started on a machine without the hypervisor, it halts and performs no processing. In addition to that, users leverage enclave attestation, e.g., Intel SGX attestation [91], to get a signed measurement that the correct codebase has been loaded to the trusted hardware.

### 4.6.4 Query submission and verification

Users write their queries in a subset of SQL that supports selections, projections, joins, and group-by aggregations. Users can supply arbitrary predicates, but they cannot run arbitrary user-defined functions. User submit queries to the optimizer, which is outside Hermetic's TCB and can reside either at the client or at the server. The optimizer then prepares a query plan to be executed by the runtime.

As explained in Section 4.5, query plans are annotated with the sensitivity of each relational operator, as well as with the epsilon to be used to add noise to the inter-

mediate results. Since the optimizer is not trusted, these privacy parameters have to be verified before the plan is executed: Hermetic has to check that the sensitivities are correct, and that the total $\varepsilon$ annotations do not exceed the privacy budgets. Sensitivities are verified by computing them from scratch based on the query plan, and comparing them against the ones attached to the incoming plan.

### 4.6.5 Query execution

If a plan is verified to be correct, it proceeds to be executed by the runtime. Before execution starts, the privacy budget is decreased based on the epsilons in the plan, and the runtime generates the Laplace noise which determines the number of fake records to pad intermediate results with. To execute a query plan, the Hermetic runtime sends all the individual operators of the plan to different Hermetic worker nodes, which in turn use the appropriate operators from Section 4.4 to perform the computation.

## 4.7 Implementation

To confirm our design and measure the performance implications of our approach, we implemented a prototype of Hermetic, which we describe next.

### 4.7.1 Hermetic hypervisor

We based the Hermetic hypervisor on Trustvisor [115], a compact extensible hypervisor that has been formally verified [159]. To support the functionality needed for establishing an OEE, we extended Trustvisor with two hypercalls, named `LockCore` and `UnlockCore`, respectively. The `LockCore` hypercall performs the following actions: (1) it checks that hardware hyperthreading and prefetching are disabled (these can be done by checking the number of logical and physical cores using `CPUID`, and using techniques from [160]), (2) it disables interrupts and preemption (3) it disables the `RDMSR` instruction for non-privileged instructions to prevent snooping on

| Data | $l_{L1}$ | $l_{L3}$ | $l^*_{L1}$ | $l^*_{L3}$ | $\hat{l}_{L1}$ | $\hat{l}_{L3}$ |
|---|---|---|---|---|---|---|
| Random | 4 | 34 | 0.68 | 3.34 | 0.74 | 5.0 |
| Ordered | | | 0.6693 | 3.8032 | | |
| Reverse | | | 0.6664 | 4.263 | | |

Table 4.2: L1 hit and miss latencies for `merge-sort`, as reported by Intel's specifications ($l_{L1}$, $l_{L3}$), and as measured on different datasets ($l^*_{L1}$, $l^*_{L3}$). The last columns show the values we used in our model. All values are in cycles.

package-level performance counters of the Hermetic core, (4) it flushes all cache lines (with `WBINVD`), (5) it uses CAT to assign a part of the LLC exclusively to the core running Hermetic (by writing to several model-specific registers [98]), and (6) it returns control to the Hermetic runtime that called the hypercall. After calling `LockCore`, the Hermetic runtime can proceed with the remaining steps required to establish an OEE (Section 4.7.2). `UnlockCore` reverts actions (2) to (5) in the reverse order. Overall, we modified 300 SLoC within the Trustvisor code.

### 4.7.2 Oblivious execution environment

**Memory-access obliviousness:** To ensure that all memory accesses from within the OEE are served from the cache, we disabled hardware prefetching, as discussed in Section 4.7.1, and we pre-loaded all data and instructions to the cache. For the former we used the `prefetcht0` instruction, which instructs the CPU to keep the data cached, and we performed dummy writes to prevent leakage through the cache coherence protocol. For the latter, we adjusted our algorithms so that they could be run in a "shortcut mode" that exercises all the code (and thus loads it into the icache) but does not touch any actual data. We also carefully aligned all buffers in memory to avoid cache collisions.

**Timing obliviousness:** To prevent the data-dependent memory accesses inside OEE from causing variations in the execution time, we pad the execution time to a safe upper bound. In principle, this could be done by adding up the worst-case latencies of all instructions and the LLC hit latency $l_{L3}$ for all memory accesses, but this would

be wildly conservative. To get a tighter bound, we carefully analyzed the two algorithms (mergesort and linear-merge) that Hermetic needs to run inside the OEE; fortunately, because of the way these algorithms access the data, it is easy to prove for many memory accesses that they will be served from the L1 cache, which allows us to substitute the L1 hit latency $l_{L1}$ for $l_{L3}$ in these cases. However, due to the superscalar execution in modern CPUs, the resulting bound is still 10x larger than the actual execution time. To further improve the bound, we performed a large number of experiments in which we measured the L1 hit and miss rates using the CPU's performance counters, and we used regression to learn *effective* L1 and L3 hit latencies $l^*_{L1}$ and $l^*_{L3}$. Table 4.2 shows the effective latencies we estimated for mergesort, as well as those of the specification. Since we could not be sure that we have observed the worst case in our experiments, we added generous safety margins to obtain bounds $\hat{l}_{L1}$ and $\hat{l}_{L3}$, which our prototype uses to determine how much to pad the execution time. The resulting times are roughly twice the actual execution times, and they were never exceeded in our experiments. For extra security, the bounds from the specification could be used instead, at the expense of somewhat lower performance. For more details please refer to Appendix C.1.

Hermetic's trusted codebase consists of the runtime, which had 3,995 SLoC in our prototype, and the trusted hypervisor, which had 14,095 SLoC. The former seems small enough for formal verification, and the latter has, in fact, been formally verified [159] prior to our modifications. (We have not yet updated the proof, but it should not be difficult.) Notice that the hypervisor would no longer be necessary with a future revision of SGX that natively supports OEEs.

## 4.8 Evaluation

Next, we report results from our experimental evaluation of Hermetic. Our experiments try to answer the following questions: (1) Does Hermetic's OEE satisfy the security properties outlined in Section 4.3? (2) What is the overhead of time padding

| Relation | Rows | Multiplicities |
|---|---|---|
| Trips | $10^7$ | m(cid)=32, m(location)=1019 |
| Customers | $4 \cdot 10^6$ | m(cid)=1 |
| Poi | $10^4$ | m(location)=500 |
| Synthetic | * | * |

Table 4.3: The schema and statistics of the relations used for our end-to-end experiments. The synthetic relation was generated for a variety of rows and multiplicities.

| Configuration | Sort | Join | MC | IC | TC | OC |
|---|---|---|---|---|---|---|
| NonOblivious | `MS` | `SMJ` | ✗ | ✗ | ✗ | ✗ |
| OblMem-NoOEE | `BS` | [13, 130] | ✓ | ✗ | ✗ | ✗ |
| Full-Padding | `BS` | `CP` | ✓ | ✗ | ✗ | ✓ |
| Hermetic I | `HS` | [13, 130]* | ✓ | ✓ | ✓ | ✗ |
| Hermetic II | `HS` | [13, 130]* | ✓ | ✓ | ✓ | ✓ |

Table 4.4: Experimental configurations and their resilience to different side channels. `MS` stands for `merge-sort`, `BS` for `batcher-sort`, `HS` for `hybrid-sort`, `CP` for cartesian product, and `SMJ` for sort-merge join. * denotes that the primitive was modified as described in Section 4.4.

inside the OEE? (3) How does having a OEE affect the performance of oblivious sorting? (4) What are the performance characteristics of Hermetic relational operators for data with different statistics? (5) Can Hermetic scale to realistic datasets and queries? (6) Can we get good performance even if we want very strong privacy guarantees?

### 4.8.1 Experimental setup

Since no existing CPU supports both SGX and CAT, we chose to experiment on an Intel Xeon E5-2600 v4 2.1GHz machine, which supports CAT, has 4 cores that share a 40 MB LLC, and features 64GB of RAM. This means that the numbers we report do not reflect any overheads due to encryption in SGX, but, as previous work [168] reports, the expected overhead of SGX in similar data-analytics applications is usually less than 2.4x. We installed the Hermetic hypervisor and Ubuntu

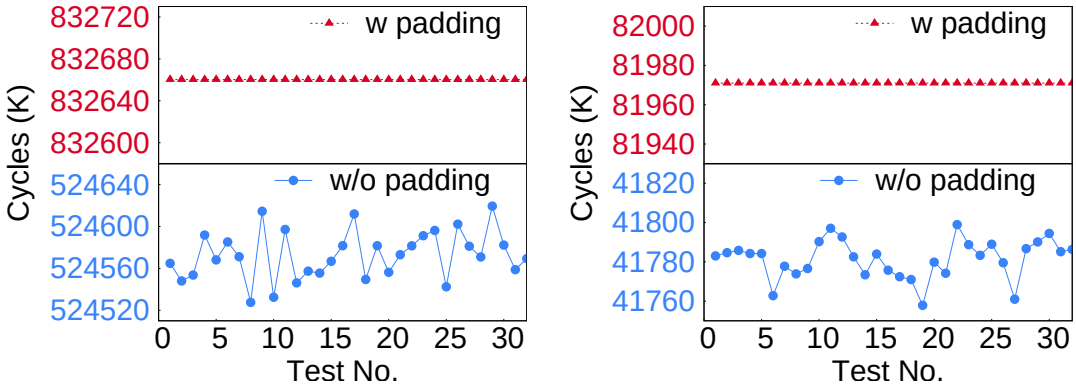
(14.04LTS) with kernel 3.2.0. We disabled hardware multi-threading, turbo-boost, and HW prefetching because they can cause timing variations.

Table 4.4 shows the different system configurations we compared, and the side channels they defend against. NonOblivious corresponds to commodity systems that take no measure against side-channels; OblMem-NoOEE uses memory-oblivious operators from previous work [13, 130], without any un-observable memory; Full-Padding performs all joins by computing the Cartesian join, which produces outputs equal to the maximum possible size of joins; and Hermetic I and Hermetic II implement the techniques described in this Chapter – the only difference being that the former does not add noise to the intermediate results.

Table 4.3 lists all the relations we used in our experiments. The `Trips` relation has 5-days-worth of records from a real-world dataset with NYC taxi-trip data [128]. This dataset has been previously used to study side-channel leakage in MapReduce [129]. Since the NYC Taxi and Limousine Commission did not release data about the `Customers` and points of interest (`Poi`) relations, we synthetically generated them. To allow for records from the `Trips` relation to be joined with the other two relations, we added a synthetic customer id column to the trips table, and we used locations from the `Trips` relation as `Poi`'s geolocations. To examine the performance of Hermetic for data with a variety of statistics, we use synthetic relations with randomly generated data in all fields, except those that control the statistics in question.

### 4.8.2 OEE security properties

Our first experiment is designed to verify that our implementation of the OEE primitive really does have stable timing and does not access main memory during the computation. To test the behavior of `merge-sort` and `linear-merge` on a wide range of input data sets, we created synthetic relations, with randomly-generated values and as many rows as needed to completely fill the available cache (187,244




(a) MergeSort (b) LinearMerge

Figure 4.4: Cycle-resolution measurements of the actual timing of `merge-sort` and `linear-merge` inside the OEE, and their padded timing, respectively.

rows of 24 bytes each).

As an initial sanity check, we used the Pin instrumentation tool [111] to record instruction traces and memory accesses; as expected, these depended only on the size of the input data. We also used Intel's performance counters to read the number of LLC misses[1] and the number of accesses that were served by the cache[2]; as expected, we did not observe any LLC misses in any of our experiments. Finally, we used `objdump` to inspect the compiler-generated code for instructions with operand-dependent timing; as expected, there were none.

Next, we used the CPU's timestamp counter to get cycle-level measurements of the execution time within the OEE, with and without padding. Figure 4.4 shows our results: as expected, the execution time without padding varied somewhat between data sets, but with padding, the difference between the maximum and minimum execution times was only 44 and 40 cycles, respectively. As Intel's benchmarking manual [68] suggests, this residual variation can be attributed to inherent inaccuracies of the timing measurement code.

[1] Using the `LONGEST_LAT_CACHE.MISS` counter.
[2] Using the `MEM_UOPS_RETIRED.ALL_LOADS` and `MEM_UOPS_RETIRED.ALL_STORES` counters.

### 4.8.3 Overhead of time padding

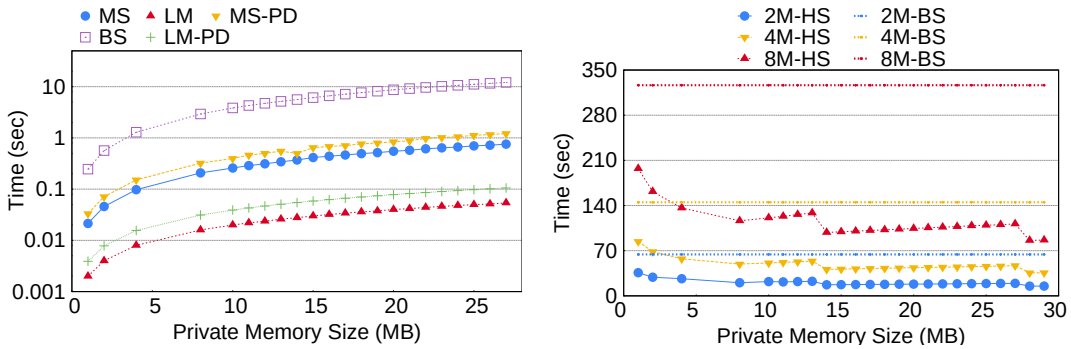
Next, we examine the overheads of padding time for `mergesort` and `linear-merge` in the OEE, and how they depend on the size of the un-observable memory.

Analogous to Section 4.8.2, we generated random data and created relations with enough rows to fill up a cache of 1MB to 27MB. On this data, we measured the time required to perform the actual computation of the two primitives, and the time spent busy-waiting to pad the execution time. We collected results across 10 runs and report the average in Figure 4.5(a). The overhead of time padding ranges between 34.2% and 61.3% for `merge-sort`, and between 95.0% and 97.9% for `linear-merge`. Even though the padding overhead of `merge-sort` is moderate, it is still about an order of magnitude faster than `batcher-sort`. This performance improvement over `batcher-sort` is enabled by having an OEE, and it is the main reason why Hermetic is more efficient than OblMem-NoOEE, even though Hermetic provides stronger guarantees.

### 4.8.4 Performance of `hybrid-sort` using OEE

Running `merge-sort` in the OEE is crucial for the good performance of `hybrid-sort`, which is the basic primitive on which all of the operators are built. To understand how these benefits depend on the size of the un-observable memory, we conducted the following experiment. We generated several synthetic relations of 2, 4, and 8 million random rows, and we measured the time required to sort them using `batcher-sort` and `hybrid-sort`. We repeated this experiment for un-observable memory sizes that ranged from 1MB to 27MB, and we report the results in Figure 4.5(b).

The results show that the larger the un-observable memory, the greater the benefits `hybrid-sort` provides. Also, as expected, the speedup compared to `batcher-sort` increases with the size of the relation. This illustrates the benefits of un-observable memory for efficient oblivious sorting.




(a) MS,LM vs BS (b) HS vs BS

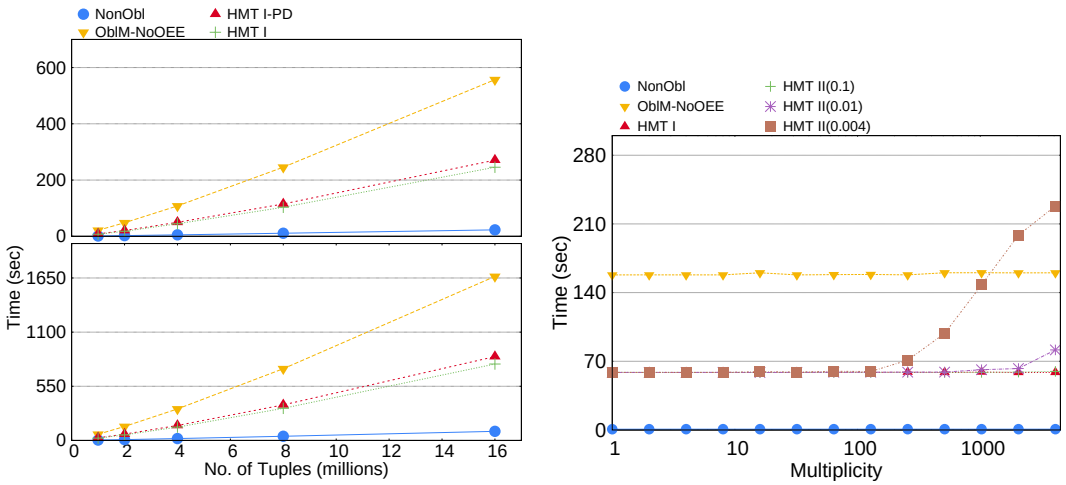
Figure 4.5: Latency of `merge-sort` (MS), `linear-merge` (LM) and `hybrid-sort` (HS) with increasing un-observable memory size, compared to the `batcher-sort` (BS).

### 4.8.5 Performance of Hermetic's relational operators

Next, we examined the performance of Hermetic's relational operators: `select`, `groupby` and `join`. For this experiment we used three simple queries ($S_1 - S_3$), whose query execution plans consist of a single relational operator. $S_1$ selects the rows of a relation that satisfy a predicate, $S_2$ groups a relation by a given column and counts how many records are per group. $S_3$ simply joins two relations. To understand the performance of the operators based on a wide range of parameters, we generated relations with different statistics (e.g., selection and join selectivities, join attribute multiplicities) and used NonOblivious, OblMem-NoOEE, and Hermetic to execute queries $S_1 - S_3$ on these relations.

Figure 4.6(a) shows the results for queries $S_1$ and $S_2$ for relations of different size. In terms of absolute performance, one can observe that Hermetic I can scale to relations with millions of records, and that the actual runtime is in the order of minutes. This is definitely slower than NonOblivious, but it seems to be a acceptable price to pay, at least for some applications that handle sensitive data. In comparison to OblMem-NoOEE, Hermetic I achieves a speedup of about 2x for all data sizes. $S_3$ displays similar behavior for increasing database sizes.

We also examined the performance of Hermetic II for query $S_3$ on relations of



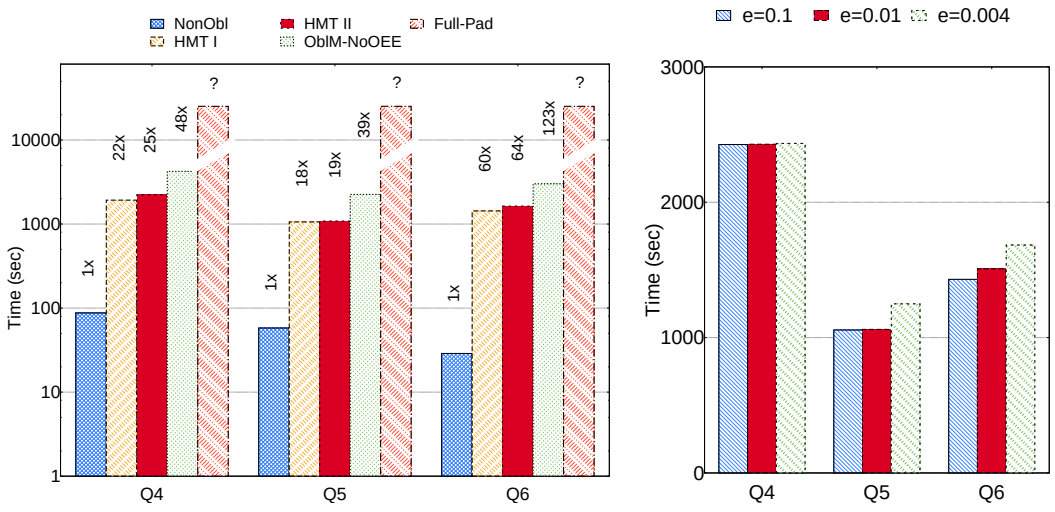

(a) $S_1$ & $S_2$ (b) $S_3$

Figure 4.6: Performance of `select` ($S_1$), `groupby` ($S_2$) and `join` ($S_3$) for different data sizes and join attribute multiplicities.

different multiplicities. The amount of noise added to the output in order to achieve the differential privacy guarantee is proportional to $s/\varepsilon$, and sensitivity $s$ is equal to the maximum multiplicity of the join attribute in the two relations. This means we expect to see a point for large multiplicity and small $\varepsilon$ where the noise become large enough to affect the performance considerably. Figure 4.6(b) shows that this threshold point is reached only for very small $\varepsilon$ and large multiplicity (around 200). However, the overhead of padding in Hermetic II is small for most combinations of multiplicity and epsilon.

### 4.8.6 Performance on realistic datasets and queries

Finally, we compared the different system configurations on complex query plans, each of which consists of at least one `select`, `groupby`, and `join` operator. To perform this experiment, we used the relations described in Table 4.3, as well as three queries that perform realistic processing on the data. $Q_4$ groups the `Customer` relation by age and counts how many customers gave a tip of at most \$10. $Q_5$ groups the points of interest relation by category, and counts the number of trips that cost less than \$15 for each category. $Q_6$ counts the number of customers that are younger




(a) Systems comparison (b) Privacy tradeoff

Figure 4.7: (a) The performance of all experimental configurations for queries $Q_4$-$Q_6$. (b) The performance of Hermetic II for $Q_4$-$Q_6$, but for plans with different privacy cost $\varepsilon$.

than 30 years old and made a trip to a hospital.

We measured the performance of all systems on these three queries, and the results are shown in Figure 4.6. Full-Padding was not able to finish, despite the fact that we left the queries running for 7 hours. This illustrates the huge cost of using full padding to combat the OC. In contrast, Hermetic II, which pads using differential privacy, has only a small overhead relative to non-padded execution (Hermetic I). This suggests that releasing a controlled amount of information about the sensitive data can lead to considerable savings in terms of performance. Also, note how `hybrid-sort` helps Hermetic be more efficient than previous oblivious processing systems (OblMem-NoOEE), *even though it offers stronger guarantees*. Overall, the performance results of Hermetic are consistent with results presented in Opaque [168], where the authors assumed the existence of an un-observable memory, and reported an average of 22x and a maximum of 63x overhead compared to NonOblivious. Even though Hermetic actually implements a real OEE and pads the OC, our results are in the same ballpark.

### 4.8.7 Performance-privacy tradeoff

The last question we answer is whether we can get good performance even if we need very strong privacy guarantees.

To answer this, we used our optimizer from Section 4.5 to identify the fastest query plans for $Q_6$ given three different bounds on the privacy consumption $\varepsilon$ of the query (0.1, 0.01, and 0.004). Figure 4.7 shows the time taken by Hermetic II to perform these optimal plans. We can see that increasing the amount of noise added to the intermediate results, i.e., decreasing epsilon the epsilon consumption of $Q_6$, has only a limited effect on performance for all queries, and the total time remains better than the time required by OblMem-NoOEE. This suggests that, even when dealing with realistic queries and data, Hermetic can achieve strong privacy guarantees by adjusting $\varepsilon$ to small values (larger noise), at only a reasonable performance cost.

## 4.9 Related Work

**Analytics on encrypted data:** In principle, privacy-preserving analytics could be achieved with fully homomorphic encryption [71] or secure multiparty computation [23], but these techniques are still orders of magnitude too slow to be practical [73, 23]. As a result, many systems use less than fully homomorphic encryption that enables some queries on encrypted data but not others. This often limits the expressiveness of the queries they support [134, 136]. In addition, some of these systems [138, 118, 141] have been shown to leak considerable information [125, 57, 81].

Alternatively, several systems [14, 17, 147] rely on TEEs or other trusted hardware. As in Hermetic, sensitive data is protected with ordinary encryption, but query processing is performed on plaintext inside enclaves. But, due to the limitations of TEEs discussed earlier, these systems do not address side channels.

**Oblivious data analytics:** Recent work has begun to focus on these side channels, but so far existing systems only address one or two, often incompletely. M2R [55] and Ohrimenko et al. [129] aim to mitigate the OC in MapReduce. Both systems reduce OC leakage, but they use ad hoc methods that still leak information about the most frequent keys. By contrast, Hermetic's OC mitigation, based on differential privacy, is more principled.

To address the MC in databases, Arasu et al. introduce a set of data-oblivious algorithms for relational operators, some based on sorting networks [13]. Ohrimenko et al. extend this set with oblivious machine learning algorithms [130]. Hermetic enhances these algorithms by making them resistant to IC, TC, and OC leakage and by speeding them up significantly using an OEE.

Opaque by Zheng et al. [168] is the system most similar to Hermetic. It combines TEEs, oblivious relational operators, and a query planner. Interestingly, Zheng et al. also recognize that performance gains are possible when small data-dependent computations can be performed entirely in the CPU cache (i.e., an OEE).[3] But, unlike Hermetic, they not describe how to realize an OEE. Furthermore, Opaque does not mitigate the IC or TC, and its mitigation for the OC relies on adding padding up to a bound determined a priori. Choosing this bound would be difficult for users: one that is too low risks privacy whereas one that is too high creates high overhead. By contrast, Hermetic computes noise automatically.

**Mitigating side channels:** It is well known that SGX does not handle most side channels [49], and recent work has already exploited several of them. These include side channels due to cache timing [42], the BTB [106], and page faults [162].

Many techniques have been proposed to mitigate so-called digital side channels that can be monitored by malware running on the target system. These include both new hardware designs and code transformations. For example, T-SGX uses transactional memory to let enclaves to detect malicious page fault monitoring [150].

[3] In fact, Zheng et al. report performance results assuming the existence of an OEE with 8MB of memory.

CATalyst uses Intel's CAT to mitigate cross-VM LLC side channels [109]. Raccoon is a compiler that rewrites programs to eliminate data-dependent branches [142]. Its techniques inspire the manual code modifications that Hermetic uses to mitigate the IC and TC. Hermetic also uses `libfixedtimefixedpoint` [12], a math library that replaces Intel's native floating point instructions, which suffer from data-dependent timing.

Hermetic cannot fully mitigate physical side channels, such as power analysis [99] or electromagnetic emanations [101] because the underlying CPU does not guarantee that its instructions are data-independent with respect to them.[4] However, these channels are most often exploited to infer a program's instruction trace. Thus, by making queries' instruction traces data-independent, Hermetic likely reduces these channels' effectiveness.

Oblivious RAMs [76, 155, 154] can eliminate memory access pattern leakage in arbitrary programs, but they suffer from poor performance in practice [142]. Moreover, ORAMs only hide the addresses being accessed, not the number of accesses, which could itself leak information [168].

## 4.10 Conclusion

In this Chapter, we have presented a principled approach to closing the four most critical side channels: memory, instructions, timing, and output size. Our approach relies on a new primitive, hardware-assisted oblivious execution environments, as well as a number of new oblivious operators and a novel privacy-aware query planner. We have presented a system called Hermetic that uses this approach; our experimental evaluation shows that Hermetic is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.

[4] e.g., Intel claims that AES-NI is resilient to digital side-channels, but does not mention others [149].

# 5

# Discussion

Even though all the systems presented in Chapters 2 to 4 are designed to perform computations on sensitive data with privacy guarantees, they have different properties that make them applicable in different situations. This Chapter discusses the differences of the three systems, in order to help interested readers decide which of them is more appropriate for a particular use case. The three systems are compared in terms of (1) their threat model, (2) the kind of computation they support, and (3) the security guarantees they provide.

## 5.1 Threat model

Selecting an appropriate threat model is very important for the design of a system that seeks to provide security and privacy guarantees. Undoubtedly, it would be great if all systems could provide the strongest available security guarantees, for instance, perfect security against multiple colluding malicious adversaries. However, this is rarely possible because of the computational cost such solutions involve.

| System | Cryptographic assumptions | Adversarial behavior | Additional assumptions |
|---|---|---|---|
| Seabed | ✓ | HbC | Single data source |
| DStress | ✓ | HbC | Up to $k$ untrusted parties |
| Hermetic | ✓ | Malicious | Trusted hardware |

Table 5.1: A comparison of the threat models of Seabed, DStress, and Hermetic.

To overcome this problem, system designers often make simplifying assumptions about the capabilities and behavior of the adversaries. The idea is that, when the system defends against weaker adversaries, the designer can use weaker, but more efficient, cryptographic schemes and optimizations. Of course, this means that users should be very careful to use these systems only if the simplifying assumptions make sense in their specific use case; for example, a system that implements access control using a single four-digit passcode, would not be appropriate to protect the banking account of a billionaire.

Table 5.1 summarizes the different assumptions made by Seabed, DStress, and Hermetic, about the number of colluding adversaries the system can defend against, their behavior during system operation, and their computational capabilities.

Not surprisingly, all three systems make the standard cryptographic assumption that the adversary does not have the computational power to break the cryptographic schemes used. This is generally considered to be acceptable - adversaries need to use machines to mount their attacks, so it is reasonable to assume that they are bounded by the limitations of current technology. In practice, this implies that the designer should carefully chose the appropriate security parameters (i.e., secret key length) for any encryption scheme used in the system.

Seabed and DStress assume that the behavior of the adversary will be Honest-but-Curious (HbC), i.e., the adversary will try to learn as much as possible, but will not actively corrupt the execution of the system protocols. On the contrary, Hermetic assumes that the adversary will try to subvert the system using arbitrarily malicious ways. Clearly, systems that defend against malicious adversaries provide

stronger security, but this comes at either an increased performance cost, or making additional assumptions that limit the capabilities of the adversary (e.g., Hermetic assumes CPUs are trusted). Therefore, the HbC model is sometimes used to achieve practical performance. As always, users should make sure that the HbC assumption makes sense in their use case. This usually means that they should examine whether adversaries would have the incentives to behave maliciously. Malicious behavior is usually much easier to detect than simple eavesdropping, so it is reasonable to assume that entities like a heavily regulated bank, or a curious cloud administrator, will try to remain under the radar.

Finally, all three systems make additional assumptions regarding the number of parties that can act adverserially in the system. Seabed assumes that data and queries originate from parties that trust each other and share a secret key. This might make sense in database outsourcing scenarios, but it is clearly not true when an analytics application involves data from multiple distrustful parties.

DStress follows a different approach: any party in the system can be compromised, but in total up to $k$ parties can collude in their attempts to extract sensitive information from the system. $k$ is a tunable parameter, and, if users are willing to pay the increased performance cost, DStress can be configured with an arbitrarily large value of $k$. Therefore, the decision whether this assumption is reasonable is equivalent to correctly choosing the $k$ parameter: assuming no more than 20 banks will collude might be reasonable, but someone could argue that assuming no more than 20 Facebook users will collude is absurd.

Hermetic can protect the sensitive data no matter the number of colluding adversaries (given at least one is legitimate). Of course, as mentioned earlier, this comes at the cost of making the assumption that the CPU package of each system machine cannot be compromised. This assumption may not hold against very powerful adversaries with the resources and expertise to extract data from a CPU; for instance, a user trying to defend against the NSA, or similar overseas agencies, should not

assume that adversaries do not know of any hardware vulnerability that can expose sensitive information.

## 5.2 Programming model

Choosing the programming model a system supports has similar tradeoffs. Ideally, we would like our systems to perform arbitrary computations on sensitive data, with strong privacy guarantees. However, this is not always possible, so system designers often settle with efficiently supporting a reduced class of computations.

Seabed and Hermetic support relational queries, and in particular, a subset of SQL. Out of this supported set, Seabed can only support an even smaller subset with strong privacy guarantees. Even though there are many analytics that cannot be computed using these restricted subsets of SQL, many useful analytics can; indeed, Section 2.6 shows that Seabed can support an entire real-world ad analytics application from Microsoft using only strong encryption techniques. Extending the supported SQL subsets is possible but not entirely straightforward: even though we believe that extending Hermetic would be mostly a matter of carefully applying the principles described in Chapter 4 to additional SQL operators, extending Seabed would be harder as it would require novel cryptographic techniques.

DStress supports vertex programs which are very powerful and can express most graph computations an analyst user would want to execute. However, DStress assumes that there is an upper bound $D$ on the number of edges that a vertex can have. Even though $D$ can be set to a value large enough to accommodate any graph, doing so will probably have adverse effects in the performance of the system, so users should choose wisely.

| System | Direct access prevention | Leakage during execution | Differential privacy guarantee |
|---|---|---|---|
| Seabed | ✓ | Columns encrypted with OPE or DET, when SPLASHE is not used. | ✗ |
| DStress | ✓ | The existence of edges may be revealed during message transfer | Adversaries cannot learn whether edges exist[1]. |
| Hermetic | ✓ | Sensitive data can be exposed through intermediate result sizes. | Adversaries cannot learn exact information about individual database records. |

Table 5.2: Security guarantees provided by Seabed, DStress, and Hermetic.

## 5.3 Security guarantees

All three systems provide basic security by preventing untrusted parties from directly accessing plaintext sensitive data. This is a very important part of any secure data processing system, but there are many ways it can be achieved - indeed, the three systems use different encryption techniques to implement this. However, large part of this work focuses on defending against more subtle ways of leakage, and Table 5.2 summarizes the sources of leakage and the privacy guarantee provided by the system.

Seabed can leak sensitive information when it executes queries that use order-preserving encryption (OPE) or deterministic encryption (DET). Even though Seabed does not limit the amount of information that can leak in these cases, it introduces techniques that can perform more queries without requiring OPE or DET. As noted in the previous Section, this approach can provide strong guarantees to some important applications, but it limits the types of analytics these security guarantees extend to.

DStress may leak sensitive information during message transfer between graph vertices. However, this leakage is carefully tracked using a privacy budget, so users get

[1] In the banking scenario, the existence of edges that represent contracts of value larger than $T$ dollars may leak (Appendix B.2.1).

a guarantee that the existence of individual edges will not be revealed to an adversary. Note, though, that this guarantee does not extend to the existence of entire vertices. This is reasonable in many applications, e.g., in the banking system, because the participation in the graph is public knowledge. Nevertheless, this might not be reasonable in scenarios where even the participation in a graph may also disclose sensitive information, e.g., in a graph that contains intelligence information about the members of criminal organization in the US.

Finally, Hermetic may leak sensitive data via side channels like the size of intermediate results, and the execution time of relational operators. As we argued in Chapter 4, approaches like full padding that completely prevent such leakage can incur performance overheads of several orders of magnitude. Therefore, Hermetic chooses to substitute the perfect hiding of full padding for the differentially-private guarantee of DP-padding. Even though this is much weaker than perfect hiding, it does provide a strong mathematical guarantee that an adversary will not be able to increase her knowledge about individual rows in the database by more than the available privacy budget. As with choices made in the design of Seabed and DStress, this sacrifice in privacy brings huge performance benefits. Again, users should make sure that protecting the privacy of individual rows makes sense in the particular scenario they are considering: e.g., protecting the rows corresponding to individual patients at a hospital might make sense, but protecting individual transactions in a person's banking account might not be enough to hide sensitive financial information.

# 6

# Conclusion

## 6.1 Summary

In this dissertation, we examined several privacy challenges that arise in applications which use sensitive data distributed across several distrustful parties. As use cases, we considered three applications: database outsourcing to untrusted servers, systemic risk computations in financial networks, and distributed database queries on machines with trusted hardware. Based on these applications, we showed how current technology is either too expensive, or it fails to prevent sensitive information leakage. To overcome these challenges, we developed novel techniques that focus on performing computations on sensitive data, while providing strong privacy guarantees and practical performance. We build three systems that use these techniques, and we evaluated their effectiveness and their performance characteristics.

In Chapter 2, we presented Seabed, a system for outsourcing analytics on sensitive data to untrusted servers. Unlike previous systems that use expensive asymmetric cryptography and deterministic encryption, which may in certain cases leak infor-

mation, Seabed uses two novel techniques called *additive symmetric homomorphic encryption (ASHE)* and *Splayed ASHE (SPLASHE)*. ASHE achieves fast aggregation on encrypted data because it uses only symmetric cryptographic primitives. SPLASHE provides stronger security because it supports a class of queries without using deterministic encryption. Our experimental evaluation showed that Seabed is efficient enough to enable analytical queries on billions of rows, and that it can support an ad analytics application from Microsoft without using weaker encryption schemes.

In Chapter 3, we described DStress, which can be used to perform graph computations on sensitive and distributed graphs with strong privacy guarantees. To achieve practical performance, DStress runs complex computations as vertex programs on graphs. To protect sensitive information during the computation, DStress runs the vertex functions in secure multi-party computation (MPC), and it leverages a *novel cryptographic protocol* to transfer MPC state between vertices, without revealing the existence of edges. Finally, DStress uses differential privacy to prevent adversaries from inferring sensitive information by looking at the result of the graph computation. The evaluation of a prototype of DStress showed that it would take just a few hours to compute the systemic risk of financial networks the size of the US banking system.

In Chapter 4, we introduced Hermetic, a system that can perform database queries on machines with trusted hardware, without leaking sensitive information via software side channels. To achieve this, we used two different techniques. First, we used a thin *trusted hypervisor* to provide strong isolation between trusted and untrusted code running on the trusted CPU, and, therefore, to preclude information leakage through shared CPU resources. Second, we replaced full padding, a technique used to hide data-dependent patterns in the timing and output sizes of relational operators, with *differentially-private padding*, which carefully adds noise to the patterns instead of hiding the completely. This lets Hermetic obtain great performance benefits, without compromising the privacy of individual database records.

Our evaluation showed that Hermetic takes just a few minutes to perform join queries on tables with millions of records, as opposed to several hours that a system using full-padding would require.

## 6.2 Future work

The three systems presented in this dissertation make a step forward towards the goal of performing analytics with privacy guarantees. However, this is by no means the end of the road; ideally, we would like to have privacy-preserving ways to perform any computation safely, with very strong guarantees, and performance comparable to systems that provide no security.

Encrypted database systems like Seabed have the disadvantage that they require several weaker encryption schemes to support complex functionality. Seabed used SPLASHE to provide select-aggregate queries on encrypted data without using weaker encryption, but for other important classes of queries, like joins, the system has to revert to weaker schemes. Therefore, a useful direction would be to develop new cryptographic techniques that enable such operations with strong security.

Another example of a useful research direction is to extend DStress so it can provide guarantees even when adversaries behave in a malicious way. As we explained in Section 3.3.2, DStress assumes that adversaries behave in an Honest-but-Curious (HbC) way, and this makes sense in the banking scenario because of regulations that are already in place. However, DStress is designed to perform general vertex programs, so it could be used in other domains where such regulations do not exist. To make DStress secure against malicious parties, one may think that it suffices to replace the secure multi-party protocol used in DStress with one from the cryptography literature that is secure against malicious adversaries. But this is not enough, because the message transfer protocol is not designed to defend against such adversaries.

Hermetic revealed that database query execution faces many interesting trade-

offs between privacy and performance. The planner described in Section 4.5 makes several simplifying choices, like selecting $\varepsilon_i$ for each operator from a vector of standard values (Section 4.5.3). However, there are many ways that the query optimizer could become smarter. First, since no privacy budget is consumed the second time the same operator is used on the same data, the planner could leverage this fact to optimize across several queries. It is not inconceivable that there might be some good selection of plans for the first few queries, that consume more budget, but reduce the total privacy consumption of the entire query trace. Second, the current Hermetic planner can miss some optimal solutions because it does not support a free choice for the $\varepsilon_i$ parameters. Hermetic could use more advanced optimization techniques, like LP-optimization, to choose an optimal value for $\varepsilon_i$ from the domain of real numbers. Finally, the planner uses noised statistics to estimate the cost of queries. Even though we have not studied the effects of noise on the accuracy of the planner's performance estimation formulas, we suspect that there is a tradeoff: the less noise we add, the more information we reveal, and the more accurate the performance estimation will be. We believe it would be worth-while examining these tradeoffs in detail.

# Bibliography


[1] AmpLab Big Data Benchmark, November 2016. https://amplab.cs.berkeley.edu/benchmark/. [Cited on pages 11 and 41.]

[2] PowerBI, November 2016. https://powerbi.microsoft.com/en-us/features/. [Cited on page 9.]

[3] Google Protocol Buffers, November 2016. https://developers.google.com/protocol-buffers/. [Cited on page 33.]

[4] Apache Spark, November 2016. http://spark.apache.org/. [Cited on pages 11 and 33.]

[5] Tableau Online, November 2016. http://www.tableau.com/products/cloud-bi. [Cited on pages 9 and 32.]

[6] Watson Analytics, November 2016. http://www.ibm.com/analytics/watson-analytics/us-en/. [Cited on pages 9 and 32.]

[7] Corda, May 2017. https://www.corda.net/. [Cited on page 80.]

[8] Emmanuel A Abbe, Amir E Khandani, and Andrew W Lo. Privacy-preserving methods for sharing financial risk exposures. *The American Economic Review*, 102(3):65–70, 2012. [Cited on pages 48, 51, and 52.]

[9] Daron Acemoglu, Asuman Ozdaglar, and Alireza Tahbaz-Salehi. Systemic risk and stability in

financial networks. *The American Economic Review*, 105(2):564–608, 2015. [Cited on page 169.]

[10] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *Proc. ACM SIGMOD*, 2004. [Cited on page 152.]

[11] Franklin Allen and Douglas Gale. Financial contagion. *Journal of political economy*, 108(1): 1–33, 2000. [Cited on page 169.]

[12] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proc. IEEE Security and Privacy*, 2015. [Cited on pages 96 and 115.]

[13] Arvind Arasu and Raghav Kaushik. Oblivious query processing. In *Proc. ICDT*, 2014. URL https://www.microsoft.com/en-us/research/publication/oblivious-query-processing/. [Cited on pages 7, 83, 89, 90, 93, 95, 106, 107, 114, and 175.]

[14] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *Proc. CIDR*, 2013. [Cited on pages 45 and 113.]

[15] ARM technical white paper. Building a secure system using TrustZone technology, May 2017. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. [Cited on page 14.]

[16] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proc. ACM CCS*, 2010. [Cited on page 89.]

[17] Sumeet Bajaj and Radu Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *Proc. SIGMOD*, 2011. [Cited on page 113.]

[18] Sumit Bajaj and Radu Sion. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014. [Cited on page 45.]

[19] Bank for international settlements. International regulatory framework for banks (Basel III). http://www.bis.org/bcbs/basel3.htm, April 2015. [Cited on page 70.]

[20] Michael Barbaro and Tom Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, August 2006. http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482. [Cited on page 53.]

[21] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2013. [Cited on page 64.]

[22] K. E. Batcher. Sorting networks and their applications. In *Proc. ACM AFIPS*, pages 307–314, 1968. [Cited on page 94.]

[23] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: Secure querying for federated databases. *Proc. VLDB Endowment*, 10(6):673–684, February 2017. [Cited on page 113.]

[24] Jethro Beekman. Improving cloud security using secure enclaves. Technical Report No. UCB/EECS-2016-219, University of California, Berkeley, 2016. [Cited on page 102.]

[25] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In *Proc. CRYPTO*, 2008. [Cited on page 80.]

[26] Robert M. Bell and Yehuda Koren. Lessons from the Netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2), 2007. [Cited on page 53.]

[27] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. ACM CCS*, 1993. [Cited on page 152.]

[28] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Proc. CRYPTO*, 2007. [Cited on pages 10, 13, 45, and 152.]

[29] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A system for secure multi-party computation. In *Proc. ACM CCS*, 2008. [Cited on page 79.]

[30] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. ACM STOC*, 1988. [Cited on pages 3 and 79.]

[31] Ranjita Bhagwan, Rahul Kumar, Ramachandran Ramjee, George Varghese, Surjyakanta Mohapatra, Hemanth Manoharan, and Piyush Shah. Adtributor: Revenue debugging in advertising systems. In *Proc. NSDI*, 2014. [Cited on page 10.]

[32] Shivam Bhasin and Francesco Regazzoni. A survey on hardware trojan detection techniques. In *Proc. IEEE ISCAS*, 2015. [Cited on page 45.]

[33] Dimitrios Bisias, Mark Flood, Andrew W. Lo, and Stavros Valavanis. A survey of systemic risk analytics. *Office of Financial Research Working Paper Series*, 1(1), 2012. [Cited on page 50.]

[34] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proc. ACM Innovations in Theoretical Computer Science*, 2013. [Cited on page 79.]

[35] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proc. ESORICS*. 2008. [Cited on page 52.]

[36] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *Proc. EUROCRYPT, 2009*. [Cited on pages 13, 45, and 152.]

[37] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proc. Theory of Cryptography*. Springer, 2005. [Cited on page 44.]

[38] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Proc. EUROCRYPT*, 2015. [Cited on page 153.]

[39] Rick Bookstaber, Jill Cetina, Greg Feldberg, Mark Flood, and Paul Glasserman. Stress tests to promote financial stability: Assessing progress and looking to the future. *Office of Financial Research Working Paper Series*, 1(10), 2013. [Cited on page 50.]

[40] Stephen P. Borgatti, Ajay Mehra, Daniel J. Brass, and Giuseppe Labianca. Network analysis in the social sciences. *Science*, 323(5916):892–895, 2009. URL http://science.sciencemag.org/content/323/5916/892. [Cited on page 56.]

[41] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. Technical Report 1701.00981, arXiv, 2017. [Cited on page 102.]

[42] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. Technical Report 1702.07521, arXiv, 2017. [Cited on pages 4, 83, 92, and 114.]

[43] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proc. USENIX Security*, 2010. [Cited on pages 52 and 79.]

[44] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016. URL http://dx.doi.org/10.1002/spe.2325. [Cited on page 28.]

[45] Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johannes Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, 2012. [Cited on page 79.]

[46] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. Practical order-revealing encryption with limited leakage. In *Proc. Fast Software Encryption*. Springer, 2016. [Cited on pages 25 and 153.]

[47] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *Proc. RSA*. 2012. [Cited on pages 52, 72, and 79.]

[48] Joao F Cocco, Francisco J Gomes, and Nuno C Martins. Lending relationships in the inter-bank market. *Journal of Financial Intermediation*, 18(1):24–48, 2009. [Cited on pages 57, 65, 77, and 170.]

[49] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical Report 2016/086, Cryptology ePrint Archive, 2016. [Cited on pages 14, 92, and 114.]

[50] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Transactions on Emerging Telecommunications Technologies*, 8 (5):481–490, 1997. [Cited on page 55.]

[51] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proc. Workshop on Public Key Cryptography*. Springer, 2009. [Cited on page 79.]

[52] Data Breach Today. Apple's ResearchKit: The privacy issues, November 2016. http://www.databreachtoday.com/apples-researchkit-privacy-issues-a-8018. [Cited on pages 1 and 2.]

[53] Shaun Davenport and Richard Ford. SGX: The good, the bad and the downright ugly. *Virus Bulletin*, January 7, 2014. URL https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly/. [Cited on page 14.]

[54] Peter Deutsch. RFC - DEFLATE compressed data format specification, May 1996. https://tools.ietf.org/html/rfc1951. [Cited on page 38.]

[55] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling stronger privacy in MapReduce computation. In *Proc. USENIX Security*, 2015. [Cited on pages 45 and 114.]

[56] Yevgeniy Dodis, Ilya Mironov, and Noah Stephens-Davidowitz. Message transmission with reverse firewalls – secure communication on corrupted machines. Technical Report 2015/548, Cryptology ePrint Archive, 2015. http://eprint.iacr.org/2015/548. [Cited on page 80.]

[57] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What else is revealed by order-revealing encryption? In *Proc. ACM CCS*, 2016. [Cited on page 113.]

[58] Cynthia Dwork. Differential privacy: A survey of results. In *Proc. Theory and Applications of Models of Computation*. Springer, 2008. [Cited on pages 49 and 78.]

[59] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. *Proc. EUROCRYPT*, 2006. [Cited on pages 72, 80, and 98.]

[60] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. Theory of Cryptography*. Springer, 2006. [Cited on pages 5, 49, 53, 54, 84, and 86.]

[61] Larry Eisenberg and Thomas H Noe. Systemic risk in financial systems. *Management Science*, 47(2):236–249, 2001. [Cited on pages 1, 48, and 66.]

[62] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proc. CRYPTO*, 1985. [Cited on page 55.]

[63] Matthew Elliott, Benjamin Golub, and Matthew O Jackson. Financial networks and contagion. *The American economic review*, 104(10):3115–3153, 2014. [Cited on pages 1, 48, 67, 68, 169, and 170.]

[64] John Fantuzzo and Dennis P Culhane. *Actionable intelligence: Using integrated data systems to achieve a more effective, efficient, and ethical government*. Springer, 2015. [Cited on pages 1 and 56.]

[65] Federal Deposit Insurance Corporation (FDIC). Dodd-Frank Act Stress Test. February 17, 2017; https://www.fdic.gov/regulations/reform/dfast/index.html. [Cited on page 66.]

[66] Mark Flood, Jonathan Katz, Stephen Ong, and Adam Smith. Cryptography and the economics of supervisory information: Balancing transparency and confidentiality. *Office of Financial Research Working Paper Series*, 1(11), 2013. [Cited on pages 1, 48, 49, 50, 51, 65, 66, 70, and 79.]

[67] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proc. ACM POPL*, 2013. [Cited on page 64.]

[68] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures, April 2017. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. [Cited on page 108.]

[69] Rodney Garratt and Peter Zimmerman. Does central clearing reduce counterparty risk in realistic financial networks? Technical Report 717, Federal Reserve Bank of New York Staff, March 2015. URL https://ssrn.com/abstract=2646040. [Cited on page 169.]

[70] Tingjian Ge and Stan Zdonik. Answering aggregation queries in a secure system model. In *Proc. VLDB, 2007*. [Cited on page 44.]

[71] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. ACM STOC*, 2009. [Cited on pages 2, 13, 14, 44, and 113.]

[72] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable Yao circuits. In *Proc. CRYPTO*, 2010. [Cited on page 80.]

[73] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Proc. CRYPTO*, 2012. [Cited on pages 2, 14, and 113.]

[74] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES Circuit (Updated Implementation), 2015. https://eprint.iacr.org/2012/099.pdf. [Cited on pages 2 and 44.]

[75] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM Journal on Computing*, 41(6):1673–1693, 2012. [Cited on pages 62 and 164.]

[76] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. [Cited on pages 14, 91, and 115.]

[77] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proc. STOC*, 1987. [Cited on pages 3, 54, 72, and 79.]

[78] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickso, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012. [Cited on page 47.]

[79] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proc. OSDI*, 2014. [Cited on page 47.]

[80] Jens Groth. Rerandomizable and replayable adaptive chosen ciphertext attack secure cryptosystems. In *Proc. Theory of Cryptography*. 2004. [Cited on page 80.]

[81] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. Technical Report 2016/895, Cryptology ePrint Archive, 2016. [Cited on page 113.]

[82] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. Differential privacy under fire. In *Proc. USENIX Security*, 2011. [Cited on pages 64 and 79.]

[83] Banchong Harangsri. *Query result size estimation techniques in database systems*. PhD thesis, The University of New South Wales, 1998. [Cited on page 99.]

[84] Brett Hemenway and Sanjeev Khanna. Sensitivity and computational complexity in financial networks. Technical Report 1503.07676, arXiv preprint, 2015. URL http://arxiv.org/abs/1503.07676. [Cited on pages 65, 68, 69, and 171.]

[85] Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C. Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. IEEE CSF*, 2014. [Cited on pages 54 and 87.]

[86] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proc. IEEE Security and Privacy*, 1991. [Cited on page 89.]

[87] IBM Corporation. IBM Systems cryptographic hardware products, November 2016. http://www-03.ibm.com/security/cryptocards/. [Cited on page 45.]

[88] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *Proc. USENIX Security*, 2013. [Cited on page 45.]

[89] Intel Corporation. Intel Software Guard Extensions SDK, developer reference, 2017. https://software.intel.com/en-us/sgx-sdk/documentation. [Cited on page 102.]

[90] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Proc. CRYPTO*, 2003. [Cited on page 75.]

[91] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing, March 2017. https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing. [Cited on pages 101 and 102.]

[92] James Greene, Intel Corporation. Intel trusted execution technology: White paper (April, 2017), April 2017. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf. [Cited on page 102.]

[93] Jennifer Ouellette, Gizmodo. Apple's health experiment is riddled with privacy problems, November 2016. http://gizmodo.com/apple-s-health-experiment-is-riddled-with-privacy-probl-1783878924. [Cited on pages 1 and 2.]

[94] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014. [Cited on page 10.]

[95] Myong H. Kang, Ira S. Moskowitz, and Daniel C. Lee. A network pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, 1996. [Cited on page 89.]

[96] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. *Proc. VLDB Endowment*, 4(11):1146–1157, 2011. [Cited on pages 64 and 78.]

[97] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, February 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology. [Cited on page 93.]

[98] Khang T Nguyen. Usage Models for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, February 2016. https://software.intel.com/en-us/articles/cache-allocation-technology-usage-models. [Cited on page 104.]

[99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proc. CRYPTO*, 1999. [Cited on pages 84 and 115.]

[100] Valdis E Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002. [Cited on page 56.]

[101] Markus Guenther Kuhn. *Compromising emanations: eavesdropping risks of computer displays*. PhD thesis, University of Cambridge, 2002. [Cited on pages 84 and 115.]

[102] Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In *Proc. International Workshop on Public Key Cryptography*, 2002. [Cited on page 72.]

[103] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16 (10):613–615, 1973. [Cited on pages 83 and 88.]

[104] David Lazer, Alex Pentland, Lada Adamic, Sinan Aral, Albert-László Barabási, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann, Tony Jebara, Gary King, Michael Macy, Deb Roy, and Marshall Van Alstyne. Computational social science. *Science*, 323(5915):721–723, 2009. URL http://science.sciencemag.org/content/323/5915/721. [Cited on pages 1 and 56.]

[105] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at twitter. In *Proc. VLDB, 2012*. [Cited on page 10.]

[106] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. Technical Report 1611.06952, arXiv preprint, 2016. [Cited on pages 4, 83, 92, and 114.]

[107] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015. [Cited on page 28.]

[108] Bao Li, Hongda Li, Guangwu Xu, and Haixia Xu. Efficient reduction of 1 out of n oblivious transfers in random oracle model. *IACR Cryptology ePrint Archive*, 2005:279, 2005. [Cited on page 75.]

[109] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. IEEE HPCA*, 2016. [Cited on page 115.]

[110] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proc. Uncertainty in Artificial Intelligence*, 2010. [Cited on pages 47 and 49.]

[111] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005. [Cited on page 108.]

[112] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. ACM SIGMOD*, 2010. [Cited on pages 49 and 56.]

[113] Sheri Markose. Systemic risk from global financial derivatives: A network analysis of contagion and its mitigation with super-spreader tax. Technical Report 12-282, International Monetary Fund Working Paper Series, 2012. [Cited on pages 169 and 170.]

[114] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. Technical Report 2017/048, Cryptology ePrint Archive, 2017. [Cited on page 102.]

[115] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. IEEE Security and Privacy*, 2010. [Cited on page 103.]

[116] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013. [Cited on pages 3, 14, and 45.]

[117] Frank McSherry. Privacy integrated queries. In *Proc. ACM SIGMOD*, 2009. [Cited on page 79.]

[118] Microsoft. MS SQL Server 2016 – Always Encrypted, November 2016. URL https://msdn.microsoft.com/en-us/library/mt163865.aspx. [Cited on pages 3 and 113.]

[119] Mike Hearn. Corda: A distributed ledger (v0.5, November, 2016). https://docs.corda.net/_static/corda-technical-whitepaper.pdf. [Cited on page 80.]

[120] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Proc. CRYPTO*, 2009. [Cited on page 57.]

[121] Arjun Narayan and Andreas Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proc. OSDI*, 2012. [Cited on pages 79 and 98.]

[122] Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen. Compute globally, act locally: Protecting federated systems from systemic threats. In *Proc. HotDep*, 2014. [Cited on page 79.]

[123] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE Security and Privacy*, 2008. [Cited on page 53.]

[124] National information center of the Federal Reserve System. Insured U.S. chartered commercial banks that have consolidated assets of $300 million or more. http://www.federalreserve.gov/releases/lbr/current/default.htm, September 2016. [Cited on page 76.]

[125] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proc. ACM CCS*, 2015. [Cited on pages 4, 10, 13, 14, 18, 45, 113, 145, and 147.]

[126] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *Proc. IEEE Security and Privacy*, 2015. [Cited on page 80.]

[127] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *Proc. ACM CCS*, 2013. [Cited on page 80.]

[128] NYC Taxi & Limousine Commission. TLC trip record data, April 2017. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. [Cited on page 107.]

[129] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proc. ACM CCS*, 2015. [Cited on pages 6, 83, 107, and 114.]

[130] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016. [Cited on pages 93, 106, 107, and 114.]

[131] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proc. ACM STOC*, 1990. [Cited on page 14.]

[132] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT, 1999*. [Cited on pages 10, 13, and 44.]

[133] Antonis Papadimitriou, Min Xu, Ariel Feldman, and Andreas Haeberlen. Hermetic: Privacy-preserving distributed analytics without (most) side channels. Under submission, May 18, 2017. [Cited on pages 7 and 8.]

[134] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *Proc. OSDI*, 2016. [Cited on pages 8, 82, and 113.]

[135] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. DStress: Efficient differentially private computations on distributed data. In *Proc. ACM EuroSys*, 2017. [Cited on page 8.]

[136] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A scalable private DBMS. In *Proc. IEEE Security and Privacy*, 2014. [Cited on page 113.]

[137] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 14(2):256–276, 1984. [Cited on page 99.]

[138] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. SOSP*, 2011. [Cited on pages 3, 10, 14, 25, 44, 82, 113, and 152.]

[139] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proc. VLDB Endowment*, 7(8):637–648, 2014. [Cited on page 64.]

[140] Ananth Raghunathan, Gil Segev, and Salil P. Vadhan. Deterministic public-key encryption for adaptively chosen plaintext distributions. In *Proc. EUROCRYPT*, 2013. [Cited on page 152.]

[141] John Randolph. Encrypted BigQuery client, November 2016. URL `https://github.com/google/encrypted-bigquery-client`. [Cited on page 113.]

[142] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proc. USENIX Security*, 2015. [Cited on pages 90 and 115.]

[143] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proc. IEEE Security and Privacy*, 2014. [Cited on page 72.]

[144] Nate Raymond and Aruna Viswanatha. U.S. regulator sues 16 banks for rigging Libor rate. Reuters, `http://www.reuters.com/article/2014/03/14/us-fdic-libor-idUSBREA2D1KR20140314`, March 2014. [Cited on pages 52 and 73.]

[145] Federal Reserve. Dodd-Frank Act Stress Test 2015: Supervisory stress test methodology and results, March 2015. URL `https://www.federalreserve.gov/newsevents/pressreleases/files/bcreg20150305a1.pdf`. [Cited on page 71.]

[146] Indrajit Roy, Srinath Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, 2010. [Cited on page 79.]

[147] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. IEEE Security and Privacy*, 2015. [Cited on pages 45, 82, 86, and 113.]

[148] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979. [Cited on page 3.]

[149] Shay Gueron. Intel Advanced Encryption Standard (Intel AES) instructions set, March 2017. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set. [Cited on page 115.]

[150] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS*, 2017. [Cited on page 114.]

[151] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012. [Cited on page 45.]

[152] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems*, 2002. [Cited on page 84.]

[153] Malcolm K Sparrow. The application of network analysis to criminal intelligence: An assessment of the prospects. *Social networks*, 13(3):251–274, 1991. [Cited on page 56.]

[154] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proc. ACM CCS*, 2013. [Cited on pages 91 and 115.]

[155] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proc. ACM CCS*, 2013. [Cited on pages 91 and 115.]

[156] Roshan Sumbaly, Jay Kreps, and Sam Shah. The big data ecosystem at LinkedIn. In *Proc. ACM SIGMOD*, 2013. [Cited on page 10.]

[157] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proc. VLDB*, 2013. [Cited on pages 3, 10, 11, 14, 25, and 44.]

[158] Christian Upper. Simulation methods to assess the danger of contagion in interbank markets. *Journal of Financial Stability*, 7(3):111–125, 2011. [Cited on page 169.]

[159] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. IEEE Security and Privacy*, 2013. [Cited on pages 103 and 105.]

[160] Viswanathan, Vish. Disclosure of H/W prefetcher control on some Intel processors, September 2014. URL https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors. [Cited on page 103.]

[161] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in intel SGX enclaves. In *Proc. ESORICS*, 2016. [Cited on page 83.]

[162] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Security and Privacy*, 2015. [Cited on pages 4, 83, 92, and 114.]

[163] Andrew Yao. Protocols for secure computations. In *Proc. FOCS*, 1982. [Cited on pages 48 and 52.]

[164] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012. [Cited on page 82.]

[165] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through independence-as-a-service. In *Proc. OSDI*, 2014. [Cited on page 56.]

[166] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In *Proc. ACM CCS*, 2013. [Cited on pages 52 and 79.]

[167] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proc. ACM CCS*, pages 305–316, 2012. [Cited on pages 83 and 88.]

[168] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proc. NSDI*, 2017. [Cited on pages 82, 83, 86, 89, 106, 112, 114, and 115.]

# A

# Seabed

## A.1 Encryption Schemes

### A.1.1 Additive Symmetric Homomorphic Encryption (ASHE)

This is a symmetric encryption scheme that is additively homomorphic. We work over any additive group (for example, $\mathbb{Z}_n$, the integers mod $n$, for a positive integer $n$). The plaintext space is $\mathbb{Z}_n$. We shall make use of a pseudorandom function (PRF) $F_k$ drawn from a function family $F : \{0,1\}^\ell \times \{0,1\}^t \to \mathbb{Z}_n$. $F_k$ is a keyed function (with key $k \in \{0,1\}^\ell$). It maps strings of length $t$ to elements in $\mathbb{Z}_n$. The security of PRFs guarantee that, given oracle access to $F_k$, no probabilistic polynomial time (PPT) adversary can distinguish the outputs of $F_k$ from a truly random function (from $\{0,1\}^t \to \mathbb{Z}_n$).

### Scheme Description

The encryption scheme is stateful and must pick a unique identifier $\mathsf{id} \in \{0,1\}^t$ for every ciphertext created[1]. We now describe the three algorithms: $\mathsf{KeyGen}, \mathsf{Enc}$, and $\mathsf{Dec}$ associated with ASHE. $\lambda$ is the security parameter.

- $\mathsf{KeyGen}(1^\lambda)$: The symmetric key for the encryption scheme is $k$, a key to the PRF.
- $\mathsf{Enc}_k(m)$: To encrypt $m \in \mathbb{Z}_n$, pick a unique $\mathsf{id} \in \{0,1\}^t$. Set the ciphertext $c = (\mathsf{id}, m - F_k(\mathsf{id}) + F_k(\mathsf{id}-1))$, where $+,-$ denote group addition/subtraction respectively.
- $\mathsf{Dec}_k(c)$: To decrypt $c = (\mathsf{id}, v)$, output $m = v + F_k(\mathsf{id}) - F_k(\mathsf{id}-1)$.

We first observe that the scheme is additively homomorphic (where the ciphertext length grows with the number of additive operations performed). If $c_1 = (\mathsf{id}_1, m_1 - F_k(\mathsf{id}_1) + F_k(\mathsf{id}_1 - 1))$, and if $c_2 = (\mathsf{id}_2, m_2 - F_k(\mathsf{id}_2) + F_k(\mathsf{id}_2 - 1))$, then $c_1 + c_2 = (\mathsf{id}_1, \mathsf{id}_2, m_1 + m_2 - F_k(\mathsf{id}_1) - F_k(\mathsf{id}_2) + F_k(\mathsf{id}_1 - 1) + F_k(\mathsf{id}_2 - 1))$. An important observation is that if we add $w$ ciphertexts $c_1, \cdots, c_w$ such that $c_i$ encrypts $m_i$ with identifier $\mathsf{id}_i$ and $\mathsf{id}_{i+1} = \mathsf{id}_i + 1$, for all $1 \leq i \leq w-1$, then the resultant ciphertext has the form: $(\mathsf{id}_1, \cdots, \mathsf{id}_w, \Sigma_{i=1}^{w} m_i - F_k(\mathsf{id}_w) + F_k(\mathsf{id}_1 - 1))$. In this special case, the ciphertext size does not grow with $w$, the number of additive operations performed, as the list $\mathsf{id}_1, \cdots, \mathsf{id}_w$ can be represented compactly.

### Proof of Security

**Lemma 1.** *ASHE is a semantically secure encryption scheme.*

*Proof.* Define $F'$ to be a new function as follows: $F'_k(\mathsf{id}) = F_k(\mathsf{id}) - F_k(\mathsf{id}-1)$. We will first prove that $F'$ is a secure pseudorandom function.

[1] We can easily make the scheme stateless by picking id randomly from a large enough space, which will ensure that the value will be unique except with negligible probability.

**Claim 1.** *If $F$ is a pseudorandom function, then $F'$ is also a pseudorandom function.*

*Proof.* Given a PPT adversary $\mathscr{A}$ that can distinguish the outputs of $F'$ from random, we will construct an adversary $\mathscr{B}$ that can distinguish the outputs of $F$ from random. $\mathscr{B}$ plays the role of the challenger in the security game against adversary $\mathscr{A}$ that breaks the security of the pseudorandom function $F'$. $\mathscr{B}$ takes part in a security game to break the security of the pseudorandom function $F$ by interacting with a challenger $\mathscr{C}$.

For every query $\mathsf{id}$ made by the adversary $\mathscr{A}$, $\mathscr{B}$ responds as follows: it queries the challenger $\mathscr{C}$ with $(\mathsf{id})$ and gets as output $R_{\mathsf{id}}$ where $R_{\mathsf{id}}$ is chosen by the challenger $\mathscr{C}$ either uniformly at random or as the output of the pseudorandom function $F_k(\mathsf{id})$ for some key $k$. Then, $\mathscr{A}$ queries the challenger with $(\mathsf{id}-1)$ and gets as output $R_{(\mathsf{id}-1)}$ where again $R_{(\mathsf{id}-1)}$ is either equal to $F_k(\mathsf{id}-1)$ or chosen uniformly at random. Now, $\mathscr{A}$ sends $(R_{id} - R_{(\mathsf{id}-1)})$ as output to $\mathscr{B}$. Note that if the same query is asked again, $\mathscr{C}$ (and subsequently $\mathscr{B}$) give the same response. Observe that if the response of $\mathscr{C}$ was the output of the pseudorandom function $F$ with key $k$, then the response of $\mathscr{B}$ to $\mathscr{A}$ is the output of the function $F'$ with the same key $k$. On the other hand, if the response of $\mathscr{C}$ is chosen uniformly at random, then the response of $\mathscr{B}$ is also chosen uniformly at random. Notice that for every $\mathsf{id}$, we rely only on the fact that $F_k(\mathsf{id})$ is pseudorandom to argue that$F'_k(\mathsf{id})$ is pseudorandom. That is, the pseudorandomness of $F_k(\mathsf{id}-1)$ is not used to argue that $F'_k(\mathsf{id})$ is random. Analogously, the pseudorandomness of $F_k(\mathsf{id})$ is used only once - to argue that $F'_k(\mathsf{id})$ is pseudorandom and not to argue about the pseudorandomness of $F'_k(\mathsf{id}+1)$ as well.

Therefore, if $\mathscr{A}$ breaks the security of the pseudorandom function $F'$ by correctly guessing whether $\mathscr{B}$ responds with the output of $F'$ or random with probability $1/2+\varepsilon$ (where $\varepsilon$ is non-negligible), then $\mathscr{B}$ makes the same guess to the challenger $\mathscr{C}$ and breaks the security of the pseudorandom function $F$ with probability $1/2+\varepsilon$. □

Given Claim 1, the proof of Lemma 1 follows in a straightforward manner from the fact that for every ciphertext, we choose a unique identifier $\mathsf{id}$. That is, given

a PPT adversary $\mathscr{A}$ that can break the semantic security of the encryption scheme, we will construct a PPT adversary $\mathscr{B}$ that can distinguish the outputs of $F'$ from random (thereby contradicting Claim 1). $\mathscr{B}$ plays the role of the challenger in the security game against adversary $\mathscr{A}$ that breaks the semantic security of the encryption scheme. $\mathscr{B}$ takes part in a security game to break the security of the pseudorandom function $F'$ by interacting with a challenger $\mathscr{C}$.

For every message $m$ queried by the adversary $\mathscr{A}$, $\mathscr{B}$ chooses an identifier $\mathsf{id}$ at random (this identifier can also be chosen by adversary $\mathscr{A}$ as long as it is uniquely chosen for every ciphertext). Then, it queries the challenger with $\mathsf{id}$ and gets as a response $R_{\mathsf{id}}$ where $R_{\mathsf{id}}$ is chosen by the challenger $\mathscr{C}$ either uniformly at random or as the output of the pseudorandom function $F'_k(\mathsf{id})$ for some key $k$. $\mathscr{B}$ then sends the pair $(\mathsf{id}, m - R_{\mathsf{id}})$ to $\mathscr{A}$ as the ciphertext for message $m$. Finally, $\mathscr{A}$ sends two messages $m_0$ and $m_1$ not queried earlier. Now, $\mathscr{B}$ has to send back a ciphertext for one of them. $\mathscr{B}$ chooses an identifier $\mathsf{id}^*$ at random and queries the challenger with $\mathsf{id}^*$. It gets a response $R_{\mathsf{id}^*}$. $\mathscr{B}$ then chooses a bit $b \in \{0,1\}$ uniformly at random and sends the pair $(\mathsf{id}^*, m_b - R_{\mathsf{id}^*})$ to $\mathscr{A}$. Now, $\mathscr{A}$ has to output a bit $b'$ indicating that the ciphertext encrypts $m'_b$. If $\mathscr{A}$ outputs $b' = b$, $\mathscr{B}$ tells the challenger $\mathscr{C}$ that his responses are pseudorandom and if $\mathscr{A}$ outputs $b' \neq b$, $\mathscr{B}$ tells the challenger $\mathscr{C}$ that his responses are random. Suppose $\mathscr{A}$ guesses $b' = b$ correctly. That is, $\mathscr{A}$ breaks the semantic security of the encryption scheme with probability $(1/2 + \varepsilon)$ where $\varepsilon$ is non-negligible. Note that if the response of the challenger $R_{\mathsf{id}^*}$ was chosen uniformly at random, then no matter how powerful $\mathscr{A}$ is, he cannot guess whether $b = 0$ or $b = 1$ with probability better than $1/2$. Therefore, if the response of $\mathscr{C}$ is random, then the probability that $\mathscr{A}$ guesses $b' = b$ is $1/2$. However, if $R_{\mathsf{id}^*}$ is the output of the pseudorandom function $F'$, then $\mathscr{A}$ can guess $b' = b$ correctly with probability $1/2 + \varepsilon$. The different events are shown below:

Here, $R_{\mathsf{id}^*} \in R$ means that $R_{\mathsf{id}^*}$ is random while $R_{\mathsf{id}^*} \in PRF$ means that it is the output of the pseudorandom function $F'$. Since the output of the challenger is

| | $b' = b$ | $b' \neq b$ |
|---|---|---|
| $R_{\mathsf{id}^*} \in R$ | $1/2+\varepsilon$ | $1/2-\varepsilon$ |
| $R_{\mathsf{id}^*} \in PRF$ | $1/2$ | $1/2$ |

either the output of $F'$ or is uniformly random each with probability $1/2$, we can observe that the adversary $\mathscr{B}$ breaks the security of the pseudorandom function with probability $((1/2*1/2)+1/2*(1/2+\varepsilon)) = (1/2+\varepsilon/2)$ (where $\varepsilon/2$ is non-negligible if $\varepsilon$ is non-negligible). Thus, the encryption scheme is semantically secure. □

### A.1.2 SPLayed Additive Symmetric Homomorphic Encryption (SPLASHE)

Databases, whose columns are encrypted with deterministic encryption are subject to frequency attacks, as illustrated by Naveed *et al.* [125]. In more detail, let $\mathscr{D}$ be a dimension in the database - for example, $\mathscr{D}$ could be a column such as gender, country, or zipcode. Let $\mathscr{M}$ be a measure in the database - for example, $\mathscr{M}$ could be revenue, salary, or number of customers. Now, suppose we are likely to make queries of the form "`select SUM(revenue) where gender = Male`". One way to allow such queries to be made over encrypted data, is to encrypt the dimension `gender` using a deterministic encryption (this would map all encryptions of `male` to the same ciphertext and all encryptions of `female` to the same ciphertext), and encrypt the measure `revenue` using an additively homomorphic encryption scheme (such as ASHE described in the previous section). Then, one can filter the rows in the database by the ciphertext corresponding to `male` and then use the additively homomorphic property of the encryption scheme to compute an encrypted copy of `SUM(revenue)`. While this is a meaningful method of achieving this query, it leaks information about the dimension column, since deterministic encryption is used. For example, simply by looking at the database, a malicious attacker can tell how many entries are there with `gender = male/female`. Mapping the ciphertext to either male or female can, in many cases, be done easily as the distribution of males

and females is quite often known.

#### Scheme Description

SPLASHE is a specialized encryption method that is designed to defeat frequency attacks on dimensions in databases. Our method leaks no information on frequency counts of dimension values and achieves full semantic security. It is efficient when the number of distinct values that the dimension could take is low (or when the number of distinct frequently occurring values that the dimension takes is low). We first describe the basic version of SPLASHE (that is efficient only when the number of distinct values that the dimension could take is low) and then show an enhanced version.

**Basic SPLASHE.** The idea is as follows: we will expand both the dimension and measure columns by a factor of $d$, where $d$ denotes the number of unique values that the dimension could take. Denote the expanded columns by $\mathscr{D}_1, \cdots, \mathscr{D}_d$ and $\mathscr{M}_1, \cdots, \mathscr{M}_d$. In our example above, the dimension (gender) could take $d = 2$ values and hence we expand the gender and revenue columns to get $\mathscr{D}_1 = \mathsf{Gender}_{\text{male}}, \mathscr{D}_2 = \mathsf{Gender}_{\text{female}}$ and $\mathscr{M}_1 = \mathsf{Revenue}_{\text{male}}, \mathscr{M}_2 = \mathsf{Revenue}_{\text{female}}$. Now, suppose we want to encrypt the $t^{\text{th}}$ row that has `gender = male` and `revenue = 100`, we set $\mathsf{Gender}_{\text{male}}[t] = 1, \mathsf{Gender}_{\text{female}}[t] = 0, \mathsf{Revenue}_{\text{male}}[t] = 100, \mathsf{Revenue}_{\text{female}}[t] = 0$, and encrypt all the four columns using an additively homomorphic encryption scheme. Note that we use a semantically secure encryption scheme and hence no frequency counts are revealed here. Now, if the client wishes to execute the query "`select SUM(revenue) where gender = Male`", he can do so by simply executing the query "`select SUM(`$\mathsf{Revenue}_{\text{male}}$`)`", which results in the same value (since the value $\mathsf{Revenue}_{\text{male}} = 0$ for all rows with `gender = female`). Furthermore, if the client wishes to hide the dimension value on which the query was made, he can execute both "`select SUM(`$\mathsf{Revenue}_{\text{male}}$`)`" as well as "`select SUM(`$\mathsf{Revenue}_{\text{female}}$`)`" and decrypting whichever value he is interested in.

**Enhanced SPLASHE.** However, if the dimension was let's say "country", then the size of the database grows significantly since the number of countries is large. We now describe a technique to reduce the size blow up in such a scenario. Suppose we knew apriori that the most frequently occurring countries in the database were going to be USA and Canada. We now create 4 columns for the dimension country and 3 columns for each measure (here we consider revenue to be one measure). We get $\mathscr{D}_1 = \mathsf{Country}_{\mathrm{USA}}, \mathscr{D}_2 = \mathsf{Country}_{\mathrm{Canada}}, \mathscr{D}_3 = \mathsf{Country}_{\mathrm{others}}, \mathscr{D}_4 = \mathsf{CountryDet}$, and $\mathscr{M}_1 = \mathsf{Revenue}_{\mathrm{USA}}, \mathscr{M}_2 = \mathsf{Revenue}_{\mathrm{Canada}}, \mathscr{M}_3 = \mathsf{Revenue}_{\mathrm{others}}$. For entries with `Country = Canada` and `Country = USA`, we encrypt similar to the method described earlier in Basic SPLASHE (i.e., for the $t^{\text{th}}$ row, if `country = USA` and `revenue = 100`, we encrypt $\mathscr{D}_1[t] = 1, \mathscr{D}_2[t] = 0, \mathscr{D}_3[t] = 0, \mathscr{M}_1[t] = 100, \mathscr{M}_2[t] = 0, \mathscr{M}_3[t] = 0$, all using ASHE). For entries which have a country other than USA or Canada, we do the following: we encrypt $\mathscr{D}_1[t] = 0, \mathscr{D}_2[t] = 0, \mathscr{D}_3[t] = 1$ and the entry in the field $\mathscr{D}_4 = \mathsf{CountryDet}$ is deterministically encrypted with the name of the country. Also we encrypt $\mathscr{M}_1[t] = 0, \mathscr{M}_2[t] = 0$, and $\mathscr{M}_3[t] = 100$ (if the revenue of this row was 100). Note, that at this point, we have not specified what to encrypt in the field $\mathscr{D}_4 = \mathsf{CountryDet}$, when `country = USA/Canada`. If we place a deterministic encryption of 0 (or some other fixed value), then the adversary can a) Learn whether a particular row's country was USA/Canada or some other country; and b) Launch frequency attacks as in [125] on the more infrequently occurring countries.

In order to overcome these security vulnerabilities, we carefully choose how to encrypt these elements. First, observe that what we need is the following: for every country apart from USA and Canada, we need to deterministically encrypt that country a *fixed* number of times in $\mathscr{D}_4$, with this number being *independent* of the number of times that country actually appeared in the database. This will ensure that, even given a frequency count of every country, the adversary cannot launch any attack using the deterministically encrypted entries of the third column.

To ensure this, we will use the "dummy" entries in $\mathscr{D}_4$ corresponding to rows

that have either USA or Canada as their country. We will deterministically encrypt (other) country values in the rows where "country" = USA/Canada and choose the country to be encrypted in such a way that the frequency counts are balanced. For example, suppose we had a total of 2500 entries in the database with USA occurring 1000 times, Canada occurring 1000 times and a total of 50 other countries each occurring lesser than 50 times each. Let's consider one of these countries, say India. Suppose it occurred 30 times. Since we have a total of 50 other (other than USA/Canada) countries and 2500 entries to be deterministically encrypted in $\mathscr{D}_4$, each country has to be encrypted 50 times in $\mathscr{D}_4$. For India, out of the 50 times that we must deterministically encrypt India, the database already has `country = India` in 30 rows. Hence, we choose 20 of the rows corresponding to `country = USA/Canada` uniformly at random and corresponding to these rows, we encrypt the column $\mathscr{D}_4 = \mathsf{CountryDet}$ with the value `country = India` using deterministic encryption. Now, in order to preserve correctness of computing revenue aggregates, note that the false entries encrypting "India" would already have the value in the field $\mathsf{Revenue}_{\mathsf{others}}$ set to be an additively homomorphic encryption of 0. Therefore, when we sum up the revenue of all rows with country as India, we will be adding 0 for every false entry and this will not affect the final sum. Once the frequency counts of all countries (other than USA/Canada) have been equally balanced, if there are any remaining rows to deterministically encrypt (this would be the case, for example if there were 2510 rows in our example above), then for each of the remaining rows, we pick a country (not USA/Canada) uniformly at random, and encrypt that country deterministically in column $\mathscr{D}_4$.

Of course, this was possible in the example we outlined above, because the countries that did not get columns of their own all occurred *infrequently*, and in particular, there were enough rows with `country = USA/Canada` where we could put in dummy deterministic encryptions of other countries. In order to ensure that this condition is indeed met, we will choose the number of countries that get separate columns,

based on the apriori estimate we have of the number of times each different value of the dimension occurs in the database. More concretely, we use a parameter $t$ such that for all values of the dimension that occur more than $t$ times in the database, that value gets a separate column. For all values that occur infrequently (less than $t$ times), in the column corresponding to all these values, we encrypt enough dummy values such that each value is encrypted $t$ times. For each remaining row, we choose an infrequently occurring value uniformly at random and encrypt that value. Note that $t$ is picked suitably so as to allow for enough "dummy" entries.

#### Proof of Security

First, observe that for a given dimension (let's say country), the only column for which we have to argue security is the CountryDet column. This is because, the entries in the remaining columns are already secure by the security of the underlying additively homomorphic encryption scheme. In order to prove the security of Enhanced SPLASHE, we make a simplifying assumption that the *input/storage order* of the rows of the database are uniformly distributed (essentially, we require that the order is independent of the dimension under consideration). If this is not true of the data itself, then one can achieve this by randomly permuting the rows of the database when storing it (either truly randomly or through the use of a secure pseudorandom permutation).

**Quantifying information leakage in Enhanced SPLASHE.** We now quantify the information that is already known to the adversary. We assume that the adversary has access to the number of rows in the database - $n$, and hence this is public information. Now, consider a dimension $\mathscr{D}$ that takes $k$ unique values $v_1, \cdots, v_k$. Let us say each $v_i$ occurs $x_i$ times in the database and that $x_1, \ldots, x_k$ are sorted in non-decreasing order. Let $t$ be a threshold. That is, for each $v_i$, if $x_i > t$, we create a separate column for $v_i$. Let us say the first $j$ entries have $x_i > t$. Therefore, we will have $(j+1)$ columns in total – the first $j$ columns for each $v_i$ with $i \leq j$ and the last column for "other". Notice

that in the last column, by design, each value $v_i$ with $i > j$ will be deterministically encrypted an equal number of times. Therefore, the adversary can count how many distinct deterministic encryptions are there in this column, which is the number of infrequently occurring values $(k-j)$. Let's call $(k-j)$ as $c$. Also, just by counting, the adversary knows that there are $j$ columns for the most frequently occurring countries. We assume that we are willing to leak to the adversary the threshold $t$. The reason being, our scheme is designed such that we encrypt all values $v_i$ for $i > j$, a total of $t$ times and then for the remaining rows, we sample a value $v_i$ for $i > j$, uniformly at random.

We now analyze the condition that should be satisfied by the threshold $t$ in order for the scheme to be realizable and secure. Each infrequently occurring value will be encrypted $t$ times. Therefore, the number of dummy entries we need must be lesser than the number of dummy entries we have available (the rows corresponding to the more frequently occurring values).

$$\sum_{i=j+1}^{i=k} (t - x_i) \leq \sum_{i=1}^{j} x_i$$

Adding $\sum x_i$ for $j+1 \leq i \leq k$, to both sides, we get

$$\sum_{i=j+1}^{i=k} t \leq \sum_{i=1}^{k} x_i$$

The right hand side is $n$. The left hand side is $(t \times c)$ where $(c = k-j)$ is the number of countries that don't have a column for themselves. Therefore, the condition is $t \leq \frac{n}{c}$. To summarize, the adversary gets the following information : the number of rows $n$, the number of infrequently occurring values $c$, and the number of frequently occurring values, $j$. We remark here that we need not leak the threshold $t$ to the adversary (of course, the adversary does know that $t \leq \frac{n}{c}$.)

Security Definition. We now outline the definition of security that Enhanced SPLASHE satisfies. We shall work with the simulation-based security framework. Informally, we will require that the real distribution of the encrypted database is indistinguishable (to any probabilistically polynomial time (PPT) adversary) from the ideal distribution of a simulated encrypted database created by a simulator that knows *only* $n, c$ and $j$. This will prove that any adversary can learn only $n, c$ and $j$ from the encrypted database. More formally,

**Definition 1.** *(Simulation-Based Security) An encryption scheme* $\mathsf{E}$ *is said to be secure with respect to the simulation-based security, if, for any database* $\mathscr{D}$ *with* $n$ *rows,* $c$ *infrequently occurring dimensions such that* $t \leq (n/c)$*, and* $j$ *frequently occurring dimensions, the view of any probabilistic polynomial time adversary* $\mathscr{A}$ *in the real world (where the database is encrypted as described in Enhanced SPLASHE) is computationally indistinguishable from the view in the ideal world where a polynomial time simulator* $\mathsf{Sim}$ *produces a (simulated copy) of an encrypted database, given only public information* $n, c$ *and* $j$*. Notationally,* $\mathsf{E.Enc}(\mathscr{D}) \approx_c \mathsf{Sim}(n, c, j)$.

**Lemma 2.** *Assuming ideal security of the deterministic encryption scheme, Enhanced SPLASHE is secure with respect to the simulation-based security (as in Definition 1).*

*Proof (Sketch).* The strategy for the simulator $\mathsf{Sim}(n, c, j)$ to output a simulated copy of an encrypted database is as follows. The simulator creates $j$ columns $\mathsf{Dimension}_1, \cdots, \mathsf{Dimension}_j$ for the $j$ frequently occurring dimension values and the column $\mathsf{Dimension}_{\text{others}}$. The simulator (probabilistically) encrypts all values in these columns with zeroes. Similarly, for all measure columns $\mathsf{Measure}_1, \cdots, \mathsf{Measure}_j$, $\mathsf{Measure}_{\text{others}}$, the simulator probabilistically encrypts all values in these columns with zeroes. Now, for the column DimensionDet, note that there are $c$ values that infrequently occur. The simulator picks a key for the deterministic encryption scheme. Call this key $k_{\mathsf{S}}$. Now, for each value $i$ in the range 1 to $c$, $\mathsf{Sim}(n, c, j)$ does the following: i) Pick $\frac{n}{c}$ empty rows uniformly at random; ii) For each of these $\frac{n}{c}$

rows, deterministically encrypt the value $i$ (with key $k_S$) and store this in the dimension column. If there are any remaining empty rows, then for each remaining empty row, pick a value $1 \leq i \leq c$ at random and deterministically encrypt this value to store. Queries to the database are handled in a natural manner by the simulator.

In the ideal world, for each infrequently occurring value $v$ of the dimension, the adversary can only see $(n/c)$ rows chosen uniformly at random with the deterministic encryption of some index $i$ in the dimension column. In the real world, for each infrequently occuring value $v$ of the dimension, the adversary can only see $(n/c)$ rows with the deterministic encryption of value $v$ in the dimension column. Based on the assumption about the rows of our database, we know that these $(n/c)$ rows are distributed uniformly at random. Thus, through a reduction to the ideal security of the deterministic encryption scheme [28, 140] (which can be attained in the random oracle model [27]), the view of the adversary in the real world is computationally indistinguishable from his view in the ideal world. □

### A.1.3 Order Preserving/Revealing Encryption

Order Preserving Encryption (OPE) allows the encryption of messages, with the property that given $c_1 = \mathsf{OPEEnc}(m_1)$ and $c_2 = \mathsf{OPEEnc}(m_2)$, $c_1 < c_2$ if and only $m_1 < m_2$ (if $m_1 = m_2$, then $c_1 = c_2$). OPE was first introduced by Agrawal *et al.* [10], and it was cryptographically first defined by Boldyreva *et al.* [36]. In many OPE schemes, it is hard to quantify the information that the adversary can learn about the plaintext, given the ciphertext (and hence security is defined for specific distribution of messages, such as messages chosen uniformly at random). Popa *et al.* [138] presented an OPE scheme with ideal security – given $n$ ciphertexts $c_1, \cdots, c_n$, the only thing that the adversary can learn is the relative ordering of the plaintexts. However, the scheme of Popa *et al.* is stateful and needs to know the set of messages $m_1, \cdots, m_n$ being encrypted *all at once*. This is undesirable for many application and especially in our case while dealing with large volumes of data. The related notion of Order

Revealing Encryption (ORE) was introduced by Boneh *et al.* [38]; informally, in an ORE, there exists an algorithm Compare that takes as input two ciphertexts $c_1$ and $c_2$ and outputs which of the underlying plaintexts is greater (this algorithm is simply the comparison function in the case of OPE). Recently, Chenette *et al.* [46] presented a construction of an ORE scheme with precise quantifiable leakage. Moreover, their construction is based only on cryptographic pseudorandom functions (PRFs) and is practical. We use this construction in our system.

The ORE scheme of Chenette *et al.* for the set of $n$ bit messages, has the following leakage function $\mathscr{L}$: $\mathscr{L}(m_1,\ldots,m_k) = \{(m_i < m_j, \mathsf{ind}_{\mathsf{diff}}(m_i,m_j)) : 1 \leq i \leq j \leq t\}$. Here, $\mathsf{ind}_{\mathsf{diff}}(m_i,m_j)$ refers to the index of the most significant bit at which the two messages differ. That is, for every pair of messages, the leakage is which message is larger and which bit do they first differ at. We now briefly describe their scheme here. Fix a security parameter $\lambda$. Let $\mathsf{F} : \mathscr{K} \times ([n] \times \{0,1\}^{n-1}) \to Z_3$ be a secure pseudorandom function (PRF).

- $\mathsf{Setup}(1^\lambda)$: Choose a PRF key $k \in \mathscr{K}$ uniformly at random and set this as the secret key sk.
- $\mathsf{Encrypt}(m,\mathsf{sk})$: Let $b_1 \ldots b_n$ be the binary representation of $m$. For each $i \in [n]$, compute $u_i = (\mathsf{F}(k,(i,b_1 b_2 \ldots b_{i-1}||0^{n-i})) + b_i) \bmod 3$, where F is the PRF. The ciphertext ct is $(u_1,\ldots,u_n)$.
- $\mathsf{Compare}(\mathsf{ct}_1,\mathsf{ct}_2)$: Let $\mathsf{ct}_1 = (u_1,\ldots,u_n)$ and $\mathsf{ct}_2 = (u'_1,\ldots,u'_n)$. Find the smallest index $i$ such that $u_i \neq u'_i$. If no such $i$ exists, both messages are equal. If $u_i = (u'_i + 1) \bmod 3$, output that $\mathsf{ct}_1$ encrypts the larger message. Else, output that $\mathsf{ct}_2$ encrypts the larger message.

## A.2 Analysis of MDX queries supported by Seabed

An important question to ask is whether Seabed supports a wide range of Big Data Analytics applications. We address this question in this section. We analyzed the

main interface that these applications use at the back-end: MDX (the industry standard). From this analysis, we have determined that Seabed's support of the functionality falls into the four categories listed below. Table A.1 shows the numbers of queries that fall into these categories for MDX queries.

SUPPORT COMPLETELY ON SERVER: Seabed's encryption techniques can support operations such as computing sum, average, count, min, and max with no client-side support. This type of query is denoted by S in Table A.2.

SUPPORT WITH CLIENT PRE-PROCESSING: Seabed can also support quadratic computation necessary for more complex analytics such as variance, standard deviation, and covariance. This is supported in Seabed by the client pre-processing certain quadratic terms and uploading them (in encrypted format using ASHE) as well. This type of query is denoted by CPre in Table A.2.

SUPPORT WITH CLIENT POST-PROCESSING: All applications and APIs we studied allow users to specify arbitrary functions of data. When these functions are complex, Seabed requires support for post-processing and compute on the client-side. This is akin to how Monomi splits queries into server- and client-side components. This type of query is denoted by CPost in Table A.2.

SUPPORT WITH TWO CLIENT ROUND-TRIPS: Some queries require the client to compute an intermediate result, encrypt it, and send it back to the server. We believe Seabed can support a single client round-trip time. This type of query is denoted by 2R in Table A.2.

We now present a more detailed analysis of the individual MDX queries and how Seabed supports these queries.

| Query set | Total | Purely on Server | Client Pre-processing | Client Post-processing | Two Round-trips |
|---|---|---|---|---|---|
| MDX | 38 | 17 | 12 | 4 | 5 |

Table A.1: Aggregate details of MDX queries that Seabed supports.

| S. No | Query Type | Description | How Seabed supports it | Seabed Type |
|---|---|---|---|---|
| 1 | Aggregate | Aggregates of measures | ASHE for Sum, Count; OPE for Max, Min | S |
| 2 | Avg | Average of measures | ASHE for Sum, Count; Client does division | S |
| 3 | CalculationCurrentPass | Current calculation pass of cube | Independent of Seabed | S |
| 4 | CalculationPassValue | Returns MDX expression value after current pass | Independent of Seabed | S |
| 5 | CoalesceEmpty | Updates empty value to numeric/string | Can be done with extra counter with identity | CPre |
| 6 | Correlation | Correlation Coefficient of two series X, Y | ASHE & precomputation of XY; Client does division | CPre |
| 7 | Count(Dimensions) | Number of dimensions in cube | Independent of Seabed | S |
| 8 | Count(Hierarchy Levels) | Number of levels in hierarchy | Independent of Seabed | S |
| 9 | Count(Set) | Number of elements in a set | Using DE or SPLASHE | S |
| 10 | Count(Tuple) | Number of dimensions in tuple | Independent of Seabed | S |
| 11 | Covariance | Covariance of X, Y | Same as Correlation | CPre |
| 12 | CovarianceN | Covariance of X, Y (with division by N-1) | Same as Correlation | CPre |
| 13 | DistinctCount | Counts distinct elements | Using DE or SPLASHE | S |
| 14 | IIf | One of two values based on logical test | Two values sent back to the client | CPost |
| 15 | LinRegIntercept | Intercept in the Regression Line using Least Squares Method | Data sent back to client for every iteration | 2R |
| 16 | LinRegPoint | $y$ in the regression line | Same as LinRegIntercept | 2R |
| 17 | LinRegR2 | Coefficient of Determination | Same as LinRegIntercept | 2R |
| 18 | LinRegSlope | Slope of the regression line | Same as LinRegIntercept | 2R |
| 19 | LinRegVariance | Var. associated with regression line | Same as LinRegIntercept | 2R |
| 20 | LookupCube | MDX expression over a cube | Data sent back to client for computation | CPost |
| 21 | Max | Maximum value in set | Using OPE | S |
| 22 | Median | Median value in set | Using OPE | S |
| 23 | Min | Minimum value in set | Using OPE | S |
| 24 | Ordinal | Zero-based ordinal value | Using OPE | S |
| 25 | Predict | Value of expression over data mining model | Data sent back to client for computation | CPost |
| 26 | Rank | One-based rank of set | Using OPE | S |
| 27 | RollupChildren | Value generated by rolling up values of children | Data sent back to client for computation | CPost |
| 28 | Stddev | Standard deviation of a set $X$ | ASHE when Client uploads encrypted $X^2$ terms | CPre |
| 29 | StddevP | Std. Dev. using biased pop. formula | Same as Stddev | CPre |
| 30 | Stdev | Alias for Stddev | Same as Stddev | CPre |
| 31 | StdevP | Alias for StddevP | Same as Stddev | CPre |
| 32 | StrToValue | Value of MDX-formatted string | Independent of Seabed | S |
| 33 | Sum | Sum over a set | Using ASHE | S |
| 34 | Value | Value of a measure as a string | Independent of Seabed | S |
| 35 | Var | Variance of a set $X$ | Same as Stddev | CPre |
| 36 | Variance | Alias for Var | Same as Stddev | CPre |
| 37 | VarianceP | Alias for VarP | Same as Stddev | CPre |
| 38 | VarP | Var. using biased pop. formula | Same as Stddev | CPre |

Table A.2: Number of MDX queries that Seabed supports.

# B

# DStress

## B.1 Proof of Message Privacy

In this section we prove the message privacy property of our share transfer protocol. Before we do so, we define what a share transfer scheme is and what it means for a transfer scheme to provide message privacy.

### B.1.1 Definitions

**Share transfer scheme:** Suppose we are given two blocks $B_u$ and $B_v$ with $k+1$ nodes each. Also, suppose a secret value $V$ is secret shared among the nodes in $B_u$. A share transfer scheme $\Pi$ is specified by the following randomized algorithms Setup, Encrypt, Aggregate, RandomizeKey, Adjust, Decrypt, Recover:

Setup: takes a security parameter $l$, a block $B_v$ and a block size $k+1$. It returns a list of $k+1$ private and public key pairs $\{(SK_0, PK_0), \ldots, (SK_k, PK_k)\}$, one for each node in $B_v$.

RandomizeKeys: takes a set of $k+1$ public keys $\{PK_0,\ldots,PK_k\}$ and randomness $r$, and returns a set of re-randomized public keys $\{PK_0^r,\ldots,PK_k^r\}$.

Encrypt: takes as input $k+1$ single bit shares $\{b_0^u,\ldots,b_k^u\}$ from block $B_u$ and a set of $k+1$ re-randomized public keys $\{PK_0^r,\ldots,PK_k^r\}$ from block $B_v$. It returns $(k+1)^2$ ciphertext values $\{c_{i,j}^r\}$ for $i \in B_u$ and $j \in B_v$.

Aggregate: takes as input an aggregation function $f$ and $(k+1)^2$ ciphertexts $\{c_{i,j}^r\}$, corresponding to plaintexts $\{b_{i,j}^u\}$, for $i \in B_u$ and $j \in B_v$ . It returns $(k+1)$ ciphertexts $c_j^r$, each corresponding to the aggregation $f(b_{i,j}^r), \forall i \in B_u$.

Adjust: takes as input randomness $r$ and a set of $k+1$ ciphertexts $\{c_0^r,\ldots,c_k^r\}$ encrypted using re-randomized public keys $\{PK_0^r,\ldots,PK_k^r\}$. It returns a set of $k+1$ adjusted ciphertexts $\{c_0,\ldots,c_k\}$ which is can be decrypted using the original secret keys $\{SK_0,\ldots,SK_k\}$.

Decrypt: takes as input $(k+1)$ ciphertexts $\{c_0,\ldots,c_k\}$ and $(k+1)$ secret keys $\{SK_0,\ldots,SK_k\}$. It returns $(k+1)$ messages $\{m_j = f(b_{i,j}^r)\}$.

Recover: takes as input $(k+1)$ messages $\{m_j = f(b_{i,j}^r)\}$ and a block $B_v$. It sets $B_v$'s $(k+1)$ bit shares $b_j^v$ to zero or one, based on the actual value of message $m_j$.

**Correctness:** A transfer scheme $\Pi$ is correct if it the value secret shared in block $B_v$ at the end of the protocol is equal to the value secret shared in block $B_u$ in the beginning of the protocol. More concretely, suppose block $B_u$ initially holds shares $\{b_i^u\}$ such that $V = \oplus_{i \in B_u} b_i^u$. If $\{PK_0,\ldots,PK_k\}$ and $\{SK_0,\ldots,SK_k\}$ are the public and private keys returned by Setup, then we say that $\Pi$ is correct if the following is true:

$$\{PK_j^r\} = RandomizeKey(r, \{PK_j\})$$

$$\{c_{i,j}^r\} = Encrypt(\{b_i^u\}, \{PK_j^r\})$$

$$\{c_j^r\} = Aggregate(f, \{c_{i,j}^r\})$$

$$\{c_j\} = Adjust(r, \{c_j^r\})$$

$$\{m_j\} = Decrypt(\{c_j\}, \{SK_j\})$$

$$\{b_j^v\} = Recover(\{m_j\})$$

$$\oplus_{j \in B_v} b_j^v = V$$

**Defining the power of the adversary:** We consider honest-but-curious (HbC), probabilistic polynomial time (PPT) adversaries, that can corrupt a set $A$ of up to $k$ members from each block $B_u$ and $B_v$. We denote such adversaries as $Adv_k$.

**Message privacy definition:** Informally, a share transfer scheme provides message privacy if it is infeasible for an adversary to learn information about the transferred secret shares by participating in the transfer.

More formally, we say that *a share transfer scheme* $\Pi$ *produces indistinguishable message transcripts in the presence of an eavesdropper*, if no $Adv_k$ adversary has a non-negligible advantage against the challenger in the $\mathsf{Transfer}_{Adv_k,\Pi}(l, B_u, B_v, A, Adv_{k,keys})$: game described below. Suppose $Adv_{k,keys} = \{(SK_j, PK_j) | j \in A \cap B_v\}$ are the key pairs of nodes in $B_v$ that are controlled by the adversary. Then:

$\mathsf{Transfer}_{Adv_k,\Pi}(l, B_u, B_v, A, Adv_{k,keys})$:

**Challenger:** The challenger generates the remaining key pairs $\{(SK_j, PK_j) | j \in B_v - A\}$ and gives the adversary the private keys in $Adv_{k,keys}$ and all the public keys.

**Adversary:** $Adv_k$ selects bit shares $\{b_i^u | i \in A \cap B_u\}$ for the nodes it controls in $B_u$ and sends them to challenger.

**Challenger:** The challenger picks a random bit $b$ and sets the secret transferred value $V$ to $b$. It then applies all the functions of $\Pi$ and records all intermediate outputs. At the end, the challenger gives $Adv_k$ all the ciphertexts $\{c_{i,j}^r | i \in B_u, j \in B_v\}$, $\{c_{i,j} | i \in B_u, j \in B_v\}$ and $\{c_j | j \in B_v\}$.

**Adversary:** $Adv_k$ inspects the ciphertexts and outputs a decision bit $b'$. We say that the adversary succeeds if $b'$ is equal to $b$.

The fact that the keys of the adversary's nodes in $B_v$ are an input to the above game

reflects our HbC assumption; all keys are selected according to the prescribed protocol. Also, the game gives $Adv_k$ access to all the information (transcript of messages) it can observe by participating in a transfer.

### B.1.2 Construction

At this point we present DStress's message transfer protocol as a share transfer scheme. We denote the construction we use in 3.3.5 as DStressTransfer and we list its functions in accordance to the definitions of the previous section. For a more detailed description of how DStressTransfer works, please refer to section 3.3.5. Let $g$ be a generator of a cyclic group $G$ of prime order $q$. Then DStressTransfer is defined by the following functions:

$$Setup(l, k+1) \rightarrow \{(x_j, g^{x_j}) | j \in B_v, x_j \in \mathbb{Z}_q\}$$

$$RandomizeKey(r, \{g^{x_j} | j \in B_v\}) \rightarrow \{g^{rx_j} | j \in B_v, r \in \mathbb{Z}_q\}$$

$$Encrypt(\{b_i^u | i \in B_u\}, \{g^{rx_j} | j \in B_v\}) \rightarrow \\ \{(g^{y_{i,j}}, g^{b_i^u} g^{rx_j y_{i,j}}) | i \in B_u, j \in B_v, y_{i,j} \in \mathbb{Z}_q\}$$

$$Aggregate(sum, \{(g^{y_{i,j}}, g^{b_i^u} g^{rx_j y_{i,j}}) | i \in B_u, j \in B_v\}) \rightarrow \\ \{(g^{\sum_{i \in B_u} y_{i,j}}, g^{\sum_{i \in B_u} b_i^u + R_j} g^{rx_j \sum_{i \in B_u} y_{i,j}}) | j \in B_v, R_j \in [R_1, R_2]\}$$

$$Adjust(r, \{(g^{\sum_{i \in B_u} y_{i,j}}, g^{\sum_{i \in B_u} b_i^u + R_j} g^{rx_j \sum_{i \in B_u} y_{i,j}}) | j \in B_v\}) \rightarrow \\ \{(g^{r \sum_{i \in B_u} y_{i,j}}, g^{\sum_{i \in B_u} b_i^u + R_j} g^{rx_j \sum_{i \in B_u} y_{i,j}}) | j \in B_v\}$$

$$Decrypt(\{(g^{r\sum_{i\in B_u} y_{i,j}}, g^{\sum_{i\in B_u} b_i^u + R_j} g^{rx_j \sum_{i\in B_u} y_{i,j}}) | j \in B_v\},$$
$$\{x_j | j \in B_v\}) \rightarrow \{\sum_{i\in B_u} b_i^u + R_j | j \in B_v\}$$

$$Recover(\{\sum_{i\in B_u} b_i^u + R_j | j \in B_v\}) \rightarrow$$
$$\{if\ (\sum_{i\in B_u} b_i^u + R_j)\ mod\ 2 = 0,$$
$$then\ b_j^v = 0,\ else\ b_j^v = 1 | j \in B_v\}$$

### B.1.3 Correctness and security proofs

**Theorem 1:** DStressTransfer is a correct share transfer scheme.

**Proof:** We will prove that DStressTransfer is correct by showing that the secret value shared at the beginning of the protocol in block $B_u$ is equal to the one shared in block $B_v$ at the end of the protocol, i.e. $V_u = V_v$.

Initially, we know that $V_u$ is equal to the XOR of all shares $\{b_i^u\}$, i.e. $V_u = \oplus_i b_i^u$. Because of the way subshares are created in Encrypt, the same holds for the subshares $b_i = \oplus_j b_{i,j}$. Additionally, from the Encrypt, Aggregate and Decrypt functions of the DStressTransfer protocol, we know that the value arriving at each member $j$ of $B_v$ is equal to $\sum_i b_{i,j}^u + R_j$. Since $R_j$ is chosen to be even, this value is even if and only if $\sum_i b_{i,j}^u$ is even. But in Recover, the shares $\{b_j^v\}$ in block $B_v$ are set to 0 if and only if the received value is even. Hence, $\{b_j^v\} = 0$ if and only if $\sum_i b_{i,j}^u$ is even. This means that $\{b_j^v\}$ takes the same value as $\oplus_i b_{i,j}^u$. By definition, $V_v = \oplus_j b_j^v$, i.e. $V_v = \oplus_j \oplus_i b_{i,j}^u$. But the XOR operation is associative and commutative, and $V_v$ can be written as $V_v = \oplus_i \oplus_j b_{i,j}^u$ or equivalently $V_v = \oplus_i b_i^u = V_u$.

**Theorem 2:** If the decisional Diffie-Hellman (*DDH*) problem is hard, then DStressTransfer provides message privacy under the definition of the previous sec-

tion.

**Proof:** We assume there is an algorithm $\mathscr{A}$ that has a non-negligible advantage in the $\mathsf{Transfer}_{Adv_k,\Pi}$ game. We will construct an algorithm $\mathscr{B}$ which uses $\mathscr{A}$ to gain a non-negligible advantage in solving the DDH problem.

To solve the DDH problem, $\mathscr{B}$ takes as input $(g, g^a, g^b, T)$ and tries to guess whether $T = g^{ab}$ or $T = g^c$, for some random $c \in \mathbb{Z}_q$. Upon receipt of $(g, g^a, g^b, T)$, $\mathscr{B}$ invokes $\mathscr{A}$ as a challenger in a $\mathsf{Transfer}_{Adv_k,\Pi}$ game, with security parameter $l$, blocks $B_u = \{u_i | i = 0 \ldots k\}$ and $B_v = \{v_j | j = 0 \ldots k\}$, and a set $A$ of nodes controlled by the adversary. Without loss of generality, assume that nodes $k_1 \in B_u$ and $k_2 \in B_v$ are not controlled by the adversary. In the first step of the game, $\mathscr{B}$ chooses key pairs as prescribed, with the exception that it sets $k_2$'s public key to be $g^a$. In the next step, $\mathscr{A}$ picks its bit shares and sends them to challenger $\mathscr{B}$. Then $\mathscr{B}$ picks random bit $b$ and sets $V = b$. Moreover, it sets all the remaining bit shares in a way that makes $V = b$ if and only if share $b_{k_1,k_2} = b$. (One way to achieve this is to set all shares randomly and the final one to be the value required so that $\oplus_{i,j} b_{i,j} = V$.) Finally, $\mathscr{B}$ sets all the ciphertexts as prescribed in the game, apart from ciphertext $c^r_{k_1,k_2}$, which is set to $(g^b, g^V T^r)$. After that, $\mathscr{A}$ outputs its decision bit $b'$ and $\mathscr{B}$ guesses $T = g^{ab}$ iff $\mathscr{A}$ succeeds, i.e. $b = b'$.

We will show that $\mathscr{B}$ has a non-negligible advantage in guessing $T$. Since $k_1$ and $k_2$ are not controlled by the adversary, $\mathscr{A}$ never gets to see plaintext $b_{k_1,k_2} = V$. So the only way to find $V$ is from either $c^r_{k_1,k_2}$, $c^r_{k_2}$ or $c_{k_2}$. If $T = g^{ab}$, then $c^r_{k_1,k_2} = (g^b, g^V g^{abr})$ is a valid ciphertext of $\mathsf{Transfer}_{Adv_k,\Pi}$ and so are $c^r_{k_2}$ and $c_{k_2}$). Because the ciphertexts are valid, $\mathscr{A}$ can guess $V$ with non-negligible probability $1/2 + \varepsilon$. If $T = g^c$, then we will argue that $c^r_{k_1,k_2} = (g^b, g^V g^{cr})$ cannot possibly reveal anything about $V$, so $\mathscr{A}$ can only guess with probability $1/2$. This is indeed true, because there exist exactly two elements $c_1$ and $c_2$ for which $g^0 g^{c_1 r} = g^1 g^{c_2 r}$. Since $c$ is randomly chosen from $\mathbb{Z}_q$, $c_1$ and $c_2$ are equally likely to be chosen as $c$, with probability $1/2$. With a similar reasoning, we can show the same about $c^r_{k_2}$ and $c_{k_2}$. So, given $\mathscr{A}$ that has

non-negligible advantage in game $\mathsf{Transfer}_{Adv_k,\Pi}$, $\mathscr{B}$ has non-negligible probability

$$\begin{aligned} P(\mathscr{B}:\ succeeds) = P(T=g^{ab})P(\mathscr{B}:\ succeeds|T=g^{ab})+ \\ P(T=g^{c})P(\mathscr{B}:\ succeeds|T=g^{c}) = \\ =1/2\cdot(1/2+\varepsilon)+1/2\cdot 1/2 = 1/2+\varepsilon/2 \end{aligned}$$

to succeed in guessing whether $T=g^{ab}$ or $T=g^{c}$, which is a contradiction because we have assumed that DDH is hard.

## B.2 Edge privacy in DStress

In this section we show how our our message transfer protocol from section 3.3.5 preserves edge differential privacy. The main idea is that we can treat the information leakage during the protocol execution as the result of a query on the graph. Since we add noise from the geometric distribution to the revealed information, we can use differential privacy to track the amount of leakage and make sure it does not exceed the threshold defined by the privacy budget of the system. In the following we prove that our protocol satisfies $\varepsilon$-DP, and show how much privacy budget is used for a concrete instantiation of DStress.

**Treating message transfer as a query:** Given a graph $G$, there are many bit-share transfers during an iteration of DStress. Each bit-share transfer from block $B_i$ to $B_j$ has the potential of revealing a small amount of information about the existence of the underlying edge $(i,j)$. This leakage happens because members of $B_j$ receive the sum of the bit-shares sent from $B_i$, instead of just the XOR of the shares. We can treat every bit-share transfer as a query $Q_{(i,j)}(G)$ on the graph $G$; the query is indexed by edge $(i,j)$ because each transfer reveals information about one edge only.

**Sensitivity:** Suppose we have two graphs $G$ and $G'$ that differ on at most one edge, say edge $(i,j)$. The sensitivity of $Q_{(i,j)}$ can be easily derived. $Q_{(i,j)}$ returns the sum of bit-shares in $B_i$ and all nodes in DStress are HbC (section 3.3.2). This means their

shares can be 0 or 1, so the global sensitivity is $\Delta = max_{G,G'}|Q_{(i,j)}(G) - Q_{(i,j)}(G')| = k+1$, that is, equal to the number of nodes in a single block.

**Release mechanism:** Remember from section 3.3.5 that, during a bit transfer from $B_i$ to $B_j$, $i$ homomorphically adds noise to the transferred sum. We will denote this noising mechanism as $Mech(G, Q_{(i,j)}, \alpha)$; the mechanism can be described with the following equation:

$$Mech(G, Q_{(i,j)}, \alpha) = Q_{(i,j)}(G) + 2 \cdot Y,$$

where $Y \sim Geo(\alpha^{\frac{2}{\Delta}})$, where *Geo* is the geometric distribution as described in [75].

**Privacy guarantee:** The above mechanism provides $\varepsilon$-differential privacy. The geometric distribution has range over all integers, and is described by the following probability density function:

$$Pr[Y = d] = \frac{\alpha - 1}{\alpha + 1} \cdot \alpha^{|d|},$$

where $\alpha$ is a parameter in $(0,1)$. For discrete distributions, the traditional differential privacy definition (section 3.3) is equivalent to proving that the ratio between $Pr[Mech(G, Q_{(i,j)}, \alpha) = d]$ and $Pr[Mech(G', Q_{(i,j)}, \alpha) = d]$ lies in the interval $[\alpha, 1/\alpha]$ (see [75], section 2.1). Adding geometric noise with parameter $\alpha$ to a query with sensitivity 1 provides $\alpha$-differential privacy [75]. Here, we will show that release mechanism *Mech* provides $\alpha$-differential privacy when we sample $Y$ from the geometric distribution with parameter $\alpha^{\frac{2}{\Delta}}$. $\alpha$ corresponds to $e^{-\varepsilon}$ from the traditional definition of differential privacy, therefore $\varepsilon = -\ln \alpha$.

Consider any two graphs $G$ and $G'$ that differ in at most one edge, say $(i, j)$, and any integer $d$ from the output range of *Mech*.

$$\frac{Pr[Mech(G, Q_{(i,j)}, \alpha^{\frac{2}{\Delta}}) = d]}{Pr[Mech(G', Q_{(i,j)}, \alpha^{\frac{2}{\Delta}}) = d]} =$$

$$= \frac{Pr[Q_{(i,j)}(G) + 2 \cdot Y = d]}{Pr[Q_{(i,j)}(G') + 2 \cdot Y = d]} = \frac{Pr[Y = \frac{d - Q_{(i,j)}(G)}{2}]}{Pr[Y = \frac{d - Q_{(i,j)}(G')}{2}]} =$$

$$= \frac{\alpha^{\frac{2}{\Delta} \cdot \frac{|d - Q_{(i,j)}(G)|}{2}}}{\alpha^{\frac{2}{\Delta} \cdot \frac{|d - Q_{(i,j)}(G')|}{2}}} = \alpha^{\frac{|d - Q_{(i,j)}(G)| - |d - Q_{(i,j)}(G')|}{\Delta}}$$

which is in $[\alpha, 1/\alpha]$ for all integers $d$, and graphs $G$ and $G'$, such that $|Q_{(i,j)}(G) - Q_{(i,j)}(G')| \leq \Delta$.

**Utility:** The sums revealed during DStress execution are not intended to be outputs of the system – we just treat them as such to keep track of information leakage. Hence, we don't care about the utility of the output; in fact we would like it to be as inaccurate as possible to any eavesdropping adversary. This suggests that we could add as much noise as possible. However, there is a technical limitation regarding the amount of noise we add: the total noised sum is transferred on the exponent of an ElGamal ciphertext, and we cannot recover exponents that are too large (section 3.3). But the geometric distribution can return, albeit with exponentially small probability, arbitrarily large noise. Therefore, there is some probability that the system will not be able to recover the ElGamal exponent – we call this the *failure probability* $P_{fail}$ of the system. Suppose we define failure to be when the system cannot decrypt an ElGamal ciphertext using a lookup table of $N_l$ entries (from $-\frac{N_l}{2}$ to $\frac{N_l}{2}$), then $P_{fail}$ is equal to the probability that the geometric distribution returns a value outside those boundaries. We can compute this probability by using the formula for the sum of the first $\frac{N_l}{2}$ terms of a geometric series (scaled by two to account for both terms from 0 to $\frac{N_l}{2}$ and from 0 to $-\frac{N_l}{2}$):

$$P_{fail} = 1 - P_{success} = 1 - 2 \cdot \frac{1 - \alpha^{\frac{N_l}{2}}}{1 + \alpha} = \frac{2\alpha^{\frac{N_l}{2}} + \alpha - 1}{1 + \alpha}$$

So, if we want $P_{fail}$ to be less than once in $N_q$ transfers, we can compute the maximum

$\alpha_{max}$ we can use by solving the inequality:

$$\frac{2\alpha^{\frac{N_l}{2}} + \alpha - 1}{1+\alpha} \leq \frac{1}{N_q} \tag{B.1}$$

**Privacy budget:** To determine exactly how much of the privacy budget is used with every iteration of DStress, we need to figure out how many sums are revealed to the adversary during that iteration. Even though a great number of message transfers take place during DStress execution, an adversary, who is trying to distinguish between two graphs $G$ and $G'$ that differ in at most one edge $(i, j)$, will only be able to extract information from the transfers that happen over that edge. Over the course of an iteration, each member of $B_j$ receives $(k+1) \cdot L$ bit-share sums because each of the $(k+1)$ members of $B_i$ sends one bit-share to each member in $B_j$, for every single bit of the $L$ bits of the DStress datatype. Since block $B_j$ can have at most $k$ colluding nodes, the adversary observes a total of $k \cdot (k+1) \cdot L$ sums. Therefore, each iteration uses $k \cdot (k+1) \cdot L \cdot \alpha_{max}$ of the $\alpha$-privacy budget.

**Concrete example:** For a concrete instantiation of DStress where each block consists of $k+1 = 20$ nodes, the sensitivity of each bit-share transfer will be $\Delta = 20$. Moreover, if we have 8GB of RAM for decryption lookup tables (i.e., $8 \cdot 8589934592$ bits) and 384-bit ciphertexts, then the lookup table will have about $N_l \simeq 230$ million entries. The total number of share transfers $N_q$ in DStress is $Y \cdot R \cdot I \cdot N \cdot D \cdot L \cdot (k+1)^2$, where $k$ is the collusion parameter, $L$ is the bit-length of transferred messages, $D$ is the number of neighbors of every node, $N$ is the number of nodes in the network, $I$ is the number of iterations in a DStress run, $R$ is the number of DStress runs per year, and $Y$ is the number of years of DStress execution. Suppose we want DStress to fail once every ten years ($Y = 10$). Then, for a setting of $R = 3$, $I = 11$, $N = 1750$, $D = 100$, $L = 16$, and $k = 19$, we get that $N_q \simeq 370$ billion.

With the parameters above and an $\varepsilon$ equal to $2.34 \cdot 10^{-7}$, that is $\alpha = 0.999999766$,

every iteration would use $k \cdot (k+1) \cdot L \cdot \varepsilon = 0.0014$ of the privacy budget[1]. Since each year has $R \cdot I = 33$ iterations, DStress's message transfer protocol would use 0.0469 of the privacy budget, before it gets replenished (section 3.4.5).

### B.2.1 Merging different privacy definitions

The message transfer protocol of DStress provides edge differential privacy, no matter what kind of graph analytics we want to compute. However, in some cases, it may be necessary to compute the analytics with a different kind of privacy guarantee. For instance, the noise DStress adds to the output of the systemic risk computations from Chapter 3.4 provides dollar differential privacy, instead of edge differential privacy. In such cases, we need to reconcile the different privacy definitions and keep a single privacy budget. In this Section, we show how one can merge the two privacy definitions and understand the exact privacy guarantee that DStress provides in the systemic risk scenario.

Dollar differential privacy differs from edge differential privacy on the definition of its *neighboring relation*. A neighboring relation determines the granularity of privacy protection: an adversary cannot use the output of the computation to identify the underlying sensitive graph among a set of neighboring graphs. The two neighboring relations are as follows:

**Dollar-privacy ($D_1$):** Two graphs $G_1$ and $G_2$ are considered to be neighbors with respect to $D_1$, if and only if the graphs are identical, except for the edge values of one vertex in $G_2$, which are a reallocation of at most $T$ dollars from the corresponding edges in $G_1$.

**Edge-privacy ($D_2$):** Two graphs $G_1$ and $G_2$ are considered to be neighbors with respect to $D_2$, if and only if they differ in at most 1 edge.

Note that none of the two definitions implies the other. To see why, consider two counter-examples. First, suppose that $G_1$ has a vertex with $T$ edges, one with

[1] Note that these values of $\alpha$, $N_l$, and $N_l$ satisfy the $P_{fail}$ inequality B.1 above.

value $T$ and the rest with 0, and the same vertex in $G_2$ has $T$ edges of value 1. Then, the two graphs are neighbors based on $D_1$, but non-neighbors based on $D_2$. Second, suppose $G_1$ has an edge of value $2T$, and $G_2$ does not have that edge. Then, the two are not neighbors based on $D_1$, but they are neighbors based on $D_2$.

DStress uses both definitions: edge privacy for share transfers, and dollar privacy for the output. To correctly limit the amount of information leaked through the entire execution of DStress, we need to keep a single budget. But to do so, we need to merge the two definitions into a new definition. This definition is based on a new neighboring relation, which is the conjunction of $D_1$ and $D_2$:

**EN/EGJ-privacy ($D_3$):** Two graphs $G_1$ and $G_2$ are considered to be neighbors with respect to $D_3$, if and only if they are neighbors both with respect to $D_1$ and $D_2$, i.e., if they differ in at most one edge whose value is less or equal than $T$ dollars.

In what follows, we will show that we can deduct the epsilon values used for share transfers and the dollar-output from a single privacy budget based on $D_3$. This is possible because a mechanism that provides either edge-privacy or dollar-privacy, provides EN/EGJ privacy too. We prove this in the following lemma.

**Lemma:** Suppose mechanism $M$ provides $\varepsilon$-DP with respect to $D_1$. Then, it provides $\varepsilon$-DP with respect to $D_3$ as well.

*Proof.* Suppose two graphs $G_1$ and $G_2$ are neighbors with respect to $D_3$. We need to show that the definition of differential privacy (Chapter 3.3) holds. But, by the definition of $D_3$, if $G_1$ and $G_2$ are neighbors, they are also neighbors with respect to $D_1$. But $M$ provides $\varepsilon$-DP based on $D_1$, so the definition of differential privacy holds:

$$Pr[M(G_1) \in S] = e^{\varepsilon} \cdot Pr[M(G_2) \in S]$$

And $M$ provides $\varepsilon$-DP with respect to $D_3$ too. □

We can use similar reasoning to prove that this is true for $D_2$ too. This means that the share transfers and the systemic risk output of DStress are protected with

$\varepsilon$-DP with respect to $D_3$ and, therefore, we can keep a single privacy budget.

$D_3$ corresponds to a privacy guarantee where all individual contracts of value less or equal than $T$ dollars are hidden during the DStress computation. This is a sensible and clearly defined guarantee, which could be used to protect banks. However, readers should be reminded that this is an active research area in economics, and further research might be necessary to find out whether this privacy guarantee would be enough to facilitate collaboration between the different banks.

## B.3 Evaluating Models of Contagion

Ideally, we would be able to test DStress on a dataset of financial interlinkages between institutions. However, for the exact privacy reasons that motivate the creation of DStress, there are no publicly available datasets on interbank linkages.

This lack of data is reflected throughout the economics literature on financial contagion. For instance, [63][§5] illustrate their model of contagion using *nation*-level debt cross-holdings, consisting of just 7 Eurozone nations. In their own words, "we include this as a proof of concept, and emphasize that the crude estimates which we use for cross-holdings make this noisy enough that we do not see the conclusions as robust, but merely as illustrative of the methodology." The remaining economics literature follows one of two paths. The first type, such as [9, 11], use stylized examples to highlight how their model of contagion behaves in that setting. The second, and by far most common, use *aggregate* bank liability data. Given the informational weakness of using aggregate data, these models are less precise. [158][§2] provides a comprehensive overview of the literature in this setting.

Other work, such as [69], takes as established that the actual banking network is *not* fully connected, but instead exhibits known properties consistent with two similar but distinct models: a core-periphery model [113] and a scale-free model. In the core-periphery model, there is a densely connected but small core set of institutions that have large aggregate assets and liabilities, surrounded by a larger set of smaller

institutions that are each individually linked to the network through one or two core institutions. In the scale-free model, banks closer to the "center" have exponentially more linkages than banks farther away from the center.

The existing work shows strong evidence that there are central institutions with dense interconnections, surrounded by a large number of less institutions that are more loosely connected. However, the exact topology is not readily inferable.

To estimate the number of iterations that our contagion detection algorithms need to be run, we looked at some hypothetical scenarios based on the empirical work we discussed above. Following [48] and [113], we performed experiments to see how our algorithms performs on simulated networks with a two-tiered structure. We created a synthetic network comprising of 50 banks. The network was stylized, with a central core of 10 banks that were densely interconnected, with the remaining banks being regional banks that were linked to one or two central banks, following the structure described in [48]. The network was designed to generate two synthetic datasets: the first where a set of regional banks failed, with the shock being absorbed by the core banks, and the second, where it led to cascading failures taking down the entire core. As with [63], we emphasize that these results should be taken as speculation based on the existing economics literature, and not definitive evidence of the structure of real world banking networks. We used this network solely to estimate the number of iterations we would need to run our algorithms for.

The core-periphery structure of the banking network ensures that shocks are transmitted to the core quickly (as all peripheral banks are within a few hops of some core bank), and, due to the densely interconnected core means that shocks that hit a single core bank transmit quickly to all the core banks, and rapidly are absorbed or trigger sizable contagion effects. Since the core banks are almost fully connected, any shortfalls spread faster than linearly within the core. Since under the core-periphery model the peripheral banks are linked to some core bank within $1-2$ hops, we only require a few additional hops for a shock to transmit through

the core itself. In our simulations, we found that shocks either escalate rapidly or not at all, and are clearly visible if the shock takes down a single core bank (because a core bank's assets and liabilities are so large). Conservatively, we estimate a bound of $\log_2 n$ in the worst case where the periphery model has a binary tree form, giving us the maximal path length before a shock reaches a core bank.

We reiterate that, due to the closed-form sensitivity proof of [84], the number of iterations does not dictate the privacy cost of the queries — only the running time of the algorithm. Thus, if set conservatively, it should detect contagion scenarios in all but the most contrived of network structures, which, according to publicly available data, clearly do not exist in practice.

# C

# Hermetic

## C.1 Determining the timing bound of an OEE

The discussion in Section 4.7.2 mentions two main techniques we used to derive a safe and efficient upper bound on the time required to execute `merge-sort` and `linear-merge` in the OEE: (1) analyzing the two algorithms to count how many accesses are guaranteed to be served by the L1 cache, and (2) computing the effective cache-access latency for L1 and LLC on superscalar CPUs. Here, we elaborate on our techniques.

**Cache-hit analysis** The obvious upper bound one can derive for `merge-sort` and `linear-merge` is to count all their memory accesses $M$, assume that all of them will be served by the LLC, and multiply the number of accesses by the latency of LLC accesses from the specification, i.e., $M \cdot l_{L3}$. However, this bound is very conservative because locality of reference suggests only a small fraction of accesses is served by the LLC.

To find a more accurate upper bound, we examined the structure of the two

primitives and found a number of accesses that are certain to be served by the L1 cache. In our analysis, we assumed that the primitives are to be run on a machine with an 8-way associative cache and that local variables are memory-aligned in a way that each variable falls in a different cache set, so that they never evict each other. Our analysis yielded the following insights:

1. Both input relations are scanned linearly;
2. All accesses to local variables are served from L1;
3. In every merging iteration, one input data access is the same as the previous iteration and, hence, served from L1;
4. Memory is fetched at cache line granularity, so each L1 miss is followed by $s_{cl}/s_{tuple}$ L1 hits, where $s_{cl} = 64$ is the cache line size in bytes, and $s_{tuple}$ is the size of sorted tuples.

(1) is clearly true because of the way the two primitives work. (2) is true because of the memory alignment of local variables and the fact that there are fewer local variables than there are cache sets, so all of them can fit in L1 and no cache set uses more than one position for local variables. Moreover, since the main loops of `merge-sort` and `linear-merge` touch all local variables, at most two data locations are accessed before the local variables are touched again, and this means no data can evict a local variable in an 8-way associative cache. (3) holds because merging involves two running pointers, and only one of the pointers advances at each iteration. (4) data is aligned in memory, so each L1 miss fetches one cache line to L1, i.e., exactly $s_{cl}/s_{tuple}$ data items. Moreover, both input relations are scanned linearly, and, therefore, for each miss, the following $s_{cl}/s_{tuple} - 1$ accesses will be L1 hits.

Our analysis helped us identify $M_{L1}$, i.e., a number of accesses that are certain to be fetched from L1. Having that, we updated our timing bound model to be $M_{L1} \cdot l_{L1} + (M - M_{L1}) \cdot l_{L3}$. In practice, we found out that a large portion of accesses are certain to be fetched from L1 (about 89.06% and 79.73% of $M$ for `merge-sort`

and `linear-merge`, respectively). The bound given by our cache-hit analysis is a safe bound, in the sense that the actual execution time cannot ever exceed the estimated bound.

**Calculating effective latencies** Even though the upper bound derived by the cache-hit analysis is much better that our initial approach, it turned out that the estimated execution time was still about an order of magnitude slower than the actual execution. This can be attributed to the fact that modern processors have a highly efficient and parallel pipeline.

To better estimate the execution time of the two primitives, we decided to measure the effective latency $l^*_{L1}$ and $l^*_{L3}$ of L1 hits and misses respectively, and plug them into the formula derived by our cache-hit analysis. To measure these effective latencies we preformed several runs of the primitives for different randomly generated input data and collected a series of measurement tuples $(m^i_{L1}, m^i_{L3}, c^i)$, where $m^i_{L1}$ was the number of L1 hits and $m^i_{L3}$ was the number of L1 misses, as reported by the CPU's performance counters, and $c^i$ was the observed execution time in cycles. Given these tuples, we assumed a linear model $m^i_{L1} \cdot l^*_{L1} + m^i_{L3} \cdot l^*_{L3} = c^i$, and used regression to derive values for $l^*_{L1}$ and $l^*_{L3}$.

The above model assumes that $l^*_{L1}$ and $l^*_{L3}$ are constant across all measurements, but this is not necessarily true. In an effort to account for outlier data that cause significantly different effective latencies, we repeated the same experiment for input data that was already sorted, and data that a was in reverse order. The results (Table 4.2) showed that, even though there is some variation in the measured latencies, it is not more than 0.02 cycles for $l^*_{L1}$, and 0.9 cycles for $l^*_{L3}$.

To make sure that our regression-based timing estimate will indeed be an upper bound of real executions, we used $\hat{l}_{L1} = 0.74$ and $\hat{l}_{L3} = 5$ as effective latencies, which are upper bounds on $l^*_{L1}$ and $l^*_{L3}$, respectively. The resulting estimates were much closer to the real execution time (at about 1.96x) than the estimates we got through the cache-hit analysis alone. Experimental results showed that the estimated bounds

were never exceeded by the actual execution time, so we believe that they can be considered a safe upper bound. However, for extra security, users of DStress could specify the timing bound to be the one from the cache-hit analysis, at some performance cost.

## C.2 Oblivious primitives with fake rows

Section 4.4 briefly mentions that we had to modify the oblivious primitives from [13] to achieve two goals: (1) allow the primitives to compute the correct result on relations that have fake rows, and (2) provide an oblivious ways of adding a controlled number of fake rows to the output of certain primitives. This Section lists the modifications we had to perform.

### C.2.1 Supporting fake rows

Fake rows in DStress are denoted by their value in the `isDummy` field. Below we list all the primitives we had to modify to account for this extra field. Keep in mind that, whenever our description involves logic with some kind of branch, we take care to replace branches with the `CMOV` instruction.

**`groupid`:** This primitive groups the rows of a relation based on a set of attributes, and adds an incremental id column, whose ids get restarted for each new group. In order for this to work correctly in the face of dummy records, we need to make sure that dummy records do not get grouped with real records. To avoid this, we expand the set of grouping attributes by adding the `isDummy` attribute. The result is that real records get correct incremental and consecutive ids.

**`grsum`:** Grouping running sum is a generalization of `groupid`, and as such, we were able to make it work with dummy records by applying the same technique as above.

**`union`:** Union expands the attributes of each relation with the attributes of the other relation, minus the common attributes, fills them up with `nil` values, and then appends the rows of the second relation to the first. To make `union` work

with fake rows, we make sure the `isDummy` attribute is considered common across all relations. This means that the output of unions has a single `isDummy` attribute, and its semantics are preserved.

**`filter`:** To make filter work with fake rows, we need to make sure that user predicates select only real rows. To achieve this, we rewrite a user-supplied predicate $p$ as "(`isDummy` $= 0$) AND $p$". This is enough to guarantee that no fake rows are selected.

**`join`:** What we want for join is that real records from the one relation are joined only with real records from the other relation. To achieve this, we include the `isDummy` attribute to the set of join attributes of the join operation.

**`groupby`:** For the `groupby` primitive, we apply the same technique as for the `groupid` and `grsum` - we expand the grouping attributes with `isDummy`.

**`cartesian-product`:** Cartesian product pairs every record of one relation with every record of the other, and this happens even for fake rows. However, we need to make sure that only one instance of `isDummy` will be in the output relation, and that it will retain its semantics. To do this, we keep the `isDummy` attribute of only one of the relations, and we update its value to be 1 if both paired rows are real and 0 otherwise.

**`multiplicity` and `histogram`:** These two primitives need to return the corresponding statistics of the real records. Therefore, we make sure to (obliviously) exclude dummy records for the computation of multiplicities and histograms.

### C.2.2 Adding fake rows to the primitive outputs

To enable the introduction of fake records, we alter the primitives `filter`, `groupby`, and `join`. The `filter` primitive normally involves extending the relation with a column holding the outcome of the selection predicate, obliviously sorting the relation based on that column, and finally discarding any records which do not satisfy the predicate. To obliviously add $N$ records, we keep $N$ of the previously discarded records, making sure to mark them as fake. `groupby` involve several stages, but their

last step is selection; therefore, one can add fake rows in the same way. `join` queries involve computing the join-degree[1] of each record in the two relations. To add noise, we modify the value of join-degree: instead of the correct value, we set the join-degree of all fake records to zero, except one, whose degree is set to $N$. By the way that joins work, this is enough to eliminate all previous fake records, and create exactly $N$ new fake records.

[1] In a join between relations $R$ and $S$, the join-degree of a record in R corresponds to the number of records in $S$ whose join attribute value is the same with this row in R.