Lens: A Service-Oriented Platform for Privacy-Preserving Computation over Personal Data

Shomil Jain
UC Berkeley

Ben Hoberman
UC Berkeley

Solomon Joseph
UC Berkeley

Abstract

Google, Apple, Facebook, and other third-party services collect, store, and operate on large quantities of personal data. These companies most frequently use this data for personalization, advertising, or other financially-motivated purposes (e.g. direct sale to data brokers). Due to the current advertising-driven paradigm of personal computing, users are faced with a tradeoff: either use third-party applications that may collect, trade, operate on, and sell personal data, or don't use these applications at all.

In this paper, we propose Lens: an end-to-end application framework built on top of EGo & OpenEnclave that enables users to leverage third party applications that perform secure computation over their personal data. Lens provides an alternative to users who're seeking to leverage third-party services without trusting such services with their data. Lens protects the privacy of sensitive user-provided data and ensures the confidentiality and integrity of data-in-use through Intel SGX enclaves. Lens allows users to initialize and use third-party application modules running on trusted hardware, while empowering a larger number of untrusted parties to take advantage of large reservoirs of personal data in a privacy-preserving manner.

1 Introduction

In line with recent technological advancements and a move towards more data-driven computation, the quantity of sensitive data generated by the Internet, social media applications, healthcare products, and the like is exponentially increasing, rendering traditional methods and infrastructure obsolete [27]. The need for massive storage and computational power has accelerated the movement towards cloud computing, in which clients outsource computation and storage to a third-party compute provider instead of running their applications on their own servers.

This inclusion of a third-party compute provider has opened a new set of attack vectors involving a potentially malicious cloud administrator [37], while expanding the trusted computing base of system architecture. Clients have no method of verifying the confidentiality and integrity of their application binaries and input data, placing them at risk of succumbing to a malicious cloud provider, who has unfettered access to the applications running on cloud hardware.

In response to the prominence of big data and a shift towards cloud computing, a new subfield of computer security has been devoted to privacy-preserving computation and storage in untrusted cloud environments [30]. While some propose homomorphic encryption [19], which allows for operations on encrypted data, as a means of providing confidentiality of data in untrusted cloud environments, the inefficiency of current homomorphic encryption protocols preclude its viability [32] in most practical use cases.

However, the use of a trusted hardware mechanism to federate user software and untrusted cloud operating systems has gained recent traction; one notable example is Intel's Software Guard Extensions (SGX) [14], which provide support for enclaves - or trusted and restricted regions of memory that shield application binaries and data from privileged software access, in its CPUs. Intel SGX provides confidentiality and integrity guarantees on client application binaries and potentially sensitive data with reasonable performance tradeoffs.

A specific use case of secure cloud computation is the secure storage and analysis of personal data. In response to the conscious pursuit of user data by service providers, the European Union passed the "Right of Access" clause [15] as part of GDPR, which made it possible for a user to request a log of the entirety of their digital interactions from an arbitrary service provider. However, prior works have found that each data controller has a different procedure for retrieving the data and a different output format [12], rendering the returned data effectively purposeless beyond a display of transparency.

Combining these datasets from multiple data sources (e.g. lifestyle, work, social, health, and financial services) yields a rich dataset for third-party developers to personalize applications. However, this dataset remains untapped, most likely due to privacy concerns around making highly personal data

available to untrusted third parties. Data brokers, in fact, likely already have access to many of these metrics - but these third-party "black market" entities leverage this data primarily for advertising.

Furthermore, users should be able to derive insights from their personal data to satiate their curiosity or better their lifestyle, but this is only possible if they are able to outsource their data to a third-party compute module in a privacy-preserving manner.

In this paper, we introduce Lens, a framework for third parties to develop analytics modules on users' personal data in a privacy-preserving manner. Utilizing Intel SGX enclaves, Lens provides users and third-party applications with strong confidentiality and integrity guarantees on their data and proprietary algorithms. We provide a library with structured requirements for a third party to develop a compute module, which they can expose for clients to use on their personal data. In addition, we provide an end-to-end workflow for clients to upload/store their data, run their data against a third-party module, and query information about their data, all in a secure manner.

We built Lens on top of EGo [5], a framework for developing confidential apps in Go to be run in secure enclaves with minimal modifications to a codebase. Lens can be run in both a secure and insecure environment, depending on the users would like to optimize for speed or security. We evaluate Lens on two workloads: analysis of Meta Messenger chats and Apple Health data. We show that Lens provides relatively minimal slowdowns in processing and querying of both forms of data, while removing the need to trust either the developers of the analytics modules or the cloud compute providers. Our contributions include:

1. An end-to-end application framework for confidential computing over personal data
2. An extension to this framework enabling secure multi-party computation, where users can perform a joint computation over the synthesis of their data
3. Two example applications leveraging Lens: a confidential search index over personal data, and a confidential multi-party health leaderboard application

1.0.1 Disclaimer to the Reader

At the beginning of this term, we set out to explore the feasibility of developing applications on top of Intel SGX with no prior confidential computing experience. One of the biggest challenges we faced in the development of Lens was getting a barebones confidential application up and running, which took us several weeks of research and exploration.

In line with the techniques discussed in class, our original approach was to develop a proof-of-concept Python application and deploy it to an SGX-enabled Linux virtual environment using SCONE [9], which enables the running of unmodified application binaries in a secure Linux environment with minimal developer overhead and SGX capabilities. However, we ran into many issues with integrating SCONE into our architecture and were ultimately blocked by the fact that the functionalities of SCONE we desired were only being offered for commercial use-cases. We spent several weeks exploring confidential applications on top of Scone, but ultimately decided to pivot to a different approach.

We acknowledge that the core technical contributions of this project are not significant; rather, we take away from this project a broad understanding of the space of confidential computing and challenges with designing and implementing a confidential system from scratch. Moreover, we demonstrate our learnings through the background and future works section of this paper.

2 Background/Related Works

Privacy-preserving computation is a core component of the Lens architecture. We first discuss several approaches and recent innovations in this space here, before going more deeply into our chosen approach.

2.1 Privacy Preserving Computation

2.1.1 Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is a cryptographic protocol that allows arbitrary computations to be performed by an untrusted party over encrypted data without compromising the encryption on the data [19]. Other versions of FHE, implemented on different underlying cryptographic schemes and and for specific tasks, have improved on the performance of FHE since its inception [10,11,20,35], but ultimately, FHE remains untenable from a performance perspective [28] because our use-case demands low-latency results for queries/inference.

2.1.2 Differential Privacy

Differential privacy [16] models quantitatively the tradeoff between adding noise to a dataset and the privacy it enables for a single person whose data appears in that dataset. Subsequent advances have developed algorithms and mechanisms for applying differential privacy to different data extraction operations, including conventional statistical summaries and more complex tasks like machine learning. Differential privacy is implemented frequently in industry and even by the U.S. Government for the 2020 Census [7,8,17,18]. Microsoft has also developed PINQ, a system intended to provide a thin layer over raw datasets which exposes a query language that enforces differential privacy of the underlying data [29]. Ultimately, we disregard differential privacy methods due to the fact that our datasets are inherently tied to protecting the

privacy of single users in cloud environments, rendering the differential privacy framework less than applicable.

2.1.3 Secure Enclaves

Secure hardware enclaves allow a CPU to run normal code in a manner that provides security and privacy guarantees against the rest of the system as long as the CPU itself is trusted. A number of hardware enclave solutions exist, but Lens runs on Intel SGX enclaves supported by the EGo Go toolchain on top of Microsoft Azure VM's. We discuss secure hardware enclaves in more depth below.

2.1.4 Hybrid Systems

Hybrid systems use a combination of cryptographic techniques and hardware enclaves to enable privacy-preserving computation.

Oblix, one such hybrid system, [31] is an efficient search index that does not reveal access patterns, supports inserts and deletes, and hides results sizes of queries. Oblix leverages several data structures, known as doubly oblivious data structures, to provide the security guarantees of obfuscating data access patterns. Oblix is enabled by hardware enclaves such as Intel SGX and relies on modifications to pre-existing ORAM data structure constructions. Its optimization for a specific use-case precludes it from being applicable to Lens' model of more general computation, but there is likely room in future research for hardening Lens by use of oblivious primitives.

2.2 Hardware Enclaves

Hardware enclaves provide a method to ensure that a computer's physical owner and host OS/hypervisor don't have access to the data being used for computation. Fundamentally, hardware enclaves accomplish this by encrypting data before it leaves the CPU, while packaging and decrypting it as it is loaded back onto it. Though implementation varies, the CPU itself (including caches, registers, and internal state) is the most common security boundary, as it is generally where modern memory management units reside. Of vendors for cloud compute processors, Intel and AMD both offer hardware support for secure enclaves on their current generations of enterprise CPUs, offered as SGX [14] and SEV [25] respectively.

Prior work has raised a number of concerns related to secure enclaves. The bandwidth and latency overhead of encrypting and decrypting data as it exits and enters the CPU is unavoidable, and the cost of context switching from enclave to non-enclave mode is difficult to mitigate. These costs, along with modern CPU architectures, indicate that unmodified, system calls and interaction with the host OS are expensive, as they involve multiple context switches. Workarounds, such as those utilized by SCONE [9], allow for these costs to be avoided in large part, while careful programming and architecture choices can mitigate other portions of these costs.

Unfortunately, some hardware implementations of SGX (all except for in the newest of processors) have extreme memory limits — less than 200MiB of memory are actually usable by a process in a secure enclave. AMD's SEV and its refinements do not suffer from these same issues but also exist only on relatively new hardware and do not have software support or provide integrity guarantees (except, again, for the latest generation hardware). Furthermore, AMD mainly aims to support entire VMs as the secure enclave, which results in a huge TCB and is not suitable for our intended application. With better software support and assuasion that the increased TCB of AMD's solution is not problematic, one area of future research may be to explore the implementation of Lens on SEV, as SEV generally reports better performance with fewer design tradeoffs.

Lens uses Intel SGX [14] for its hardware enclave. SGX aims to facilitate the smallest possible code trusted computing base, measured in LoC (the entire CPU design must always be trusted, but nothing else outside of the programmer's control is), which implies a set of design choices and performance tradeoffs.

SGX relies on attestation to initialize its root of trust. When an enclave is initialized with SGX, the memory corresponding to the *in-enclave* portion of the code is attested for via a hash and signature. As long as the CPU and Intel are trusted, this ensures that the code running in the enclave *exactly* matches with what is expected. This bootstraps trust as long as the behavior of the code running in the enclave is well-understood. It also presumes that side-channel leakage does not exist, which we consider out-of-scope for this paper.

Each enclave is allocated a special region of working memory where it stores its code and data. The CPU's memory controller entirely denies access to this region of memory to any parties other than the enclave itself, which, combined with the always-encrypted nature of any data not on the CPU itself and flushing of the CPU's data on context switches, means that the data and code running in an enclave should be entirely hidden from malicious observers. In practice, there are elements of CPU state that are not properly flushed, but there is no reason that this issue isn't solvable thorugh future hardware revisions.

2.2.1 OpenEnclave

OpenEnclave [6] is a framework for developing secure enclave code on top of Intel SGX. Specifically, OpenEnclave handles three main tasks related to using secure enclaves:

1. Handling enclave setup and teardown.
2. Handling attestation.
3. Providing runtime support for in-enclave code.

Tasks one and two are handled according to the platform requirements dictated by the host OS and hardware. They are not the most relevant to the effective use of the hardware enclaves - this abstraction is one of OpenEnclave's purposes. Of more interest in point three: hardware enclaves do not on their own provide sufficient security guarantees since the host OS can't be trusted to return well-formed, non-malicious responses to system calls (see, for example, Iago attacks [13]). Since OpenEnclave is used by application developers to write software that runs in hardware enclaves, it does *not* have to do the work of ensuring that the application's syscalls are fully mediated since application code is trusted.

OpenEnclave provides this level of runtime support primarily via custom runtime libraries and a programming paradigm intended to help automatically manage application code that runs both in-enclave and on the host. OpenEnclave provides versions of musl's libc, LLVM's libc++, OpenSSL, and Mbed TLS in addition to custom OpenEnclave libraries for attestation and control flow across the enclave boundary. These facilitate programming at a reasonably high level while fully utilizing the benefits of a trusted hardware enclave.

2.2.2 EGo

EGo [5] is a set of programming tools for writing programs that use SGX enclaves in Go. EGo primarily exists as a way to allow Go applications to run on top of OpenEnclave, meaning that the high-level programming tools and paradigms supported by EGo programs is similar to those supported by OpenEnclave, just in Go instead of C/C++. EGo provides a modified Go compiler that allows Go to integrate with the OpenEnclave ecosystem by changing how Go library calls resolve to syscalls.

2.2.3 Cloud Providers

While a number of cloud compute providers are beginning to offer solutions for confidential computing, Microsoft Azure [3]'s are the most complete. Amazon AWS [4] uses in-house secure hardware instead of industry-standard features like AMD SEV or Intel SGX, which instantly renders its offerings not practical for our use cases, in which we don't want to trust cloud providers. Google's offerings [2] do include Intel SGX-supporting machines, but Azure provides the best ease-of-use.

Azure offers a few confidential compute services. It has always-encrypted database options, presumably using techniques similar to those in CryptDB [33], confidential VMs (supported by AMD's hardware-based secure VM support), and standard VMs with enclave support (either AMD or Intel). The confidential VM solution promises to run code as-is with hardware-enforced security guarantees. Unfortunately, implementation details are somewhat opaque and not open-source, which renders the attestation-supported security guarantees somewhat untrustworthy. These VMs are also still in beta, but if the transparency issues are resolved and the features brought to production, they may be promising.

We utilize a barebones VM with SGX-backed hardware. Hardware attestation means we don't have to trust any non-open-source code to achieve intended privacy guarantees. The software tools we use are agnostic to hosting provider, so for robustness or security purposes, it should be simple to migrate to another provider (or to use multiple providers simultaneously) as long as a VM with SGX support is present.

3 Design Overview

3.1 Threat Model

As in previous work [9, 36], we assume a powerful adversary who has access and control over the cloud's software stack. The attacker has root user privileges to the operating system, can modify data or communications outside of the secure enclave, and can observe memory access and system calls from the enclave to untrusted portion of memory.

As the secure enclave is part of the trusted computing base of the system, we assume that the attacker cannot compromise/attack the hardware, steal enclave keys, or interfere with remote attestation. Furthermore, availability attacks, particularly DoS, are out of scope, because an untrusted cloud provider can choose to cut off access to computational resources at any time. In addition, side-channel and timing attacks are considered out of scope, on account of the difficulty to protect against side-channel attacks in a system centered on hardware enclaves. Furthermore, we do not consider attacks stemming from access pattern leakage in scope, namely because of the limited support offered by the EGo framework in securely writing to disk and allowing outbound network requests. The limited functionality provided by EGo out-of-the-box provides some built-in privacy guarantees, but simultaneously opens the door for future directions in the development of Lens.

Under the aforementioned threat model, Lens guarantees the confidentiality and integrity of both proprietary third-party compute algorithms and personal data uploaded to the Lens application.

3.2 Assumptions

In designing an end-to-end system for secure computation over personal data, we have been guided by assumptions that offer both challenges and opportunities. We now lay out our assumptions in more detail.

- The system is built on top of the public cloud, in order to make the PoC as close to a production-scale implementation as possible. As such, the system must assume that third-party applications are capable (and willing) of running applications in third-party contexts, and must

provide application owners with guarantees to ensure proprietary secrets are not leaked to cloud providers (e.g. sensitive model weights).

- The system is capable of supporting variable-length individual datasets. We expect real-world user-provided datasets to be anywhere from a few MB (e.g. lightweight text data) to several hundred GB (e.g. images required for processing). To restrict our scope for this project, we assess systems that use text-based datasets formatted as CSV/JSON files of up to 100 MB.

- User-provided datasets are updated infrequently. We make this assumption to reduce the scope of this project, though we admit that in real-world scenarios, personal data updates at a frequency of thousands of data points per minute. In our future studies section, we discuss an architecture to integrate real-time data sources into the Lens framework, but we restrict our current proposal to write-once, read-many data downloads only.

- Application workloads may be classified into **preprocessing/training** phases and **query/inference** phrases. Based on our research, the majority of modern applications that rely on large quantities of personal data often require some form of index-building, model-training, or other preprocessing steo - as such, we build this into the architecture of our platform. We accept larger latencies for this preprocessing stage, with the intent of optimizing the query/inference stage for fast information lookup times.

- The system must be capable of scaling to hundreds of clients that simultaneously update and query data. Here, we lean on existing scalable design paradigms (e.g. autoscaling Kubernetes clusters on Azure), but we specifically note the importance of this for a real-world implementation.

- Application providers must be willing to trust us (the providers of Lens) with their application modules, and any proprietary secrets that these modules may rely on (e.g. model weights). This is a temporary requirement to reduce the scope of this project; one major area of future research is to explore a system in which application providers can deploy their code directly to Lens to bypass our current approach of porting, validating, and embedding modules into the core Lens architecture package.

- Applications (and their dependencies) on Lens must not make any outbound network requests; all processing must be done locally. This prevents information leakage from potentially malicious or compromised third-party applications.

3.3 Interface

Lens provides a simple interface for users and third-party service owners.

Users interact with Lens through HTTPS requests made to the Lens web server. Our PoC contains simple username/-password authentication, as well as a CLI to abstract away remote attestation to the server, which occurs over RA-TLS. We discuss this in more detail in Architecture.

Third-party service owners deploy services to Lens by migrating core service logic over to GoLang, and implementing the two API functions provided by the Lens framework: `preprocess` and `query`.

Service owners can perform model training, index building, or other computationally-expensive procedures in the `preprocess` method. This method also allows application owners to define which datasets their service depends on; users that attempt to initialize a module without uploading the required datasets will receive an error message. Moreover, Lens will prevent the module from accessing datasets that it doesn't explicitly state in the module requirements.

Service owners can perform model inference, search handling, or other lookup-based procedures in the `query` method.

Both methods allow application owners to receive parameters from users (e.g. the `text` of a search query) - these parameters can be used to customize the behavior and resuls of the preprocessing or query phases.

3.4 Lens' Architecture

Lens consists of a single *master* running multiple *services* and is accessed by multiple *clients*, as shown in figure 1.

Whenever users connect to the master, they do so over RA-TLS, an open-source implementation of TLS with built-in remote attestation. The connection is made directly to the web server, which is running inside of the secure enclave on the master.

In our proof-of-concept, the web server manages data storage, data processing, and the initialization and execution of third-party services; in practice, though, each of these components would be abstracted away to different services running in a confidential Kubernetes service mesh on something like Marblerun (https://github.com/edgelesssys/marblerun).

Our main contributions surround the intervention between client and third-party services to allow for secure computation on personal user data in a mutually distrusting manner. The ultimate goal of Lens is to provide a symbiotic relationship between data sources (users) and those who seek to tap into user data to generate rich insights without forfeiting user data or access to proprietary algorithms. For the purpose of this paper, our evaluation will be on two proof-of-concept modules: messaging chat search and health metrics computation.

Individuals who wants to use Lens will first authenticate with a username and password, the combination of which

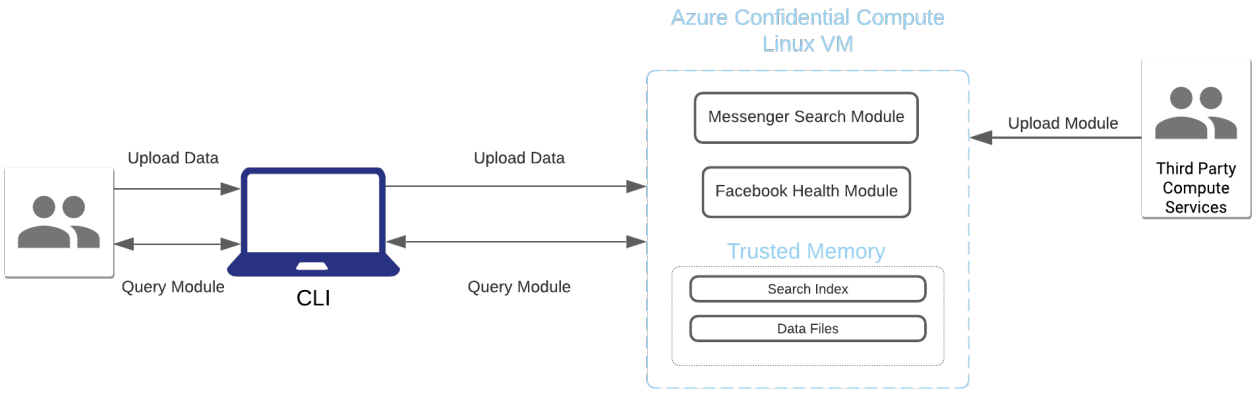


Figure 1: Lens' Architecture Overview

establishes a unique identity tied to users' personal data. Then, users interact with the command-line interface to upload and preprocess their data, the details of which are abstracted away from the user. Finally, the user can make queries, as prescribed by the module, on their personal data.

Third-party developers may develop modules for interfacing and analyzing with client data in a secure manner. Developers are required to expose a certain set of functions to plug the module into the existing infrastructure. In the current state of Lens, developers simply publish their code under a directory with a given module name, and their module is open to use by users of Lens.

3.5 RA-TLS

When the CLI connects to the server, Lens uses RA-TLS to perform remote attestation [26]. RA-TLS combines Intel SGX remote attestation with the establish of a Transport Layer Security (TLS) connection. In RA-TLS, the *attester* (a Lens master) wants to convince the *challenger* (a Lens client) that it is a genuine Lens instance running inside of an Intel SGX enclave. In RA-TLS, the TLS protocol remains unchanged, but the following steps are taken:

1. The enclave generates a new public/private keypair at startup; this is used to establish a secure channel through TLS.
2. The key is bound to the enclave by including a hash of the public key as user-data in the SGX report.
3. The server sends the quote to IAS (Intel Attestation Service) to obtain a quote verification report. Then, it creates a self-signed certificate.
4. The server sends the attestation verification report, the attestation verification report signature (signed by IAS), and the attestation report signing certificate to the challenger.
5. The challenger performs verification of the attestation report signing certificate and the enclave's identity (contained within the user data). If validation fails, the connection is aborted.

4 Implementation

4.1 Command-Line Interface

Since the command-line interface is not a significant contribution of Lens, we prioritize simplicity and straightforward design to meet functionality and security requirements for a simple proof-of-concept.

The command-line interface was written in Go using the Go library `urfave/cli` [1] for rapid development and easy configuration. The command-line interface has the following commands:

- `auth`: Retrieve and store credentials for current user. These credentials are used to determine access to previously uploaded data and authorize the querying of modules run on users' personal data.
- `upload`: Upload users' personal data to the server. This action performs a recursive zip on the user-specified data directory and executes a POST request to write the data to trusted memory. Since the HTTP request is performed via TLS, it is unnecessary to encrypt the data with a separate set of keys.
- `preprocess`: Perform preprocessing (as prescribed by the desired module) on the user data to prepare the data for querying. This action is performed after all necessary data is uploaded. The actual operations triggered by

this call are specified by the desired third-party module, which exposes a `Preprocess()` function.

- `query`: Handle arbitrary queries on preprocessed data. The *query* command serves as a wrapper around a regular HTTP request, as the client specifies the desired module, desired endpoint, and the body corresponding to the request. The CLI performs the request and returns the response back to the client.

With regards to Lens, there is ambiguity as to who interfaces with the command-line interface. While a user could make queries on their personal data to derive rich insights generated by third-party modules, the client could also be a third party who would like access to general/limited insights on the personal data; the only requirement is that the third-party is authorized to perform such queries. Lens allows for multi-party data collaboration and querying with the use of a *groupID*, attaching uploaded data items with a *groupID*.

Furthermore, note that the commands provided by the CLI are intentionally isolated to allow for better control and configurability by the user when analyzing their data. A user would be required to authenticate themselves once before they are able to upload, preprocess, and query their data an arbitrary number of times. Unintended behavior or invalid commands are gracefully handled by the command-line interface, with ample help and suggestions to simplify use.

4.2 Server

Since EGo does not offer separation between untrusted and trusted execution environments/memory, unlike other solutions [9], we choose to run the application entirely in trusted memory. In addition, EGo's lack of support for outgoing HTTP requests from applications running in trusted memory prevented us from separating data storage from the application and utilizing frequency smoothing techniques such as Pancake [21] to protect against access pattern leakage. While storing all data in trusted memory may lead to scalability issues, the use cases detailed in the paper were feasible under this storage model. In addition, we were able to attain comparable runtimes in the trusted execution environment under this model, as compared to the baseline model.

4.2.1 Authentication and Authorization

When a user authorizes with the server, they provide a username/password combination to ensure that they are capable of only receiving their own data. Optionally, they can be assigned a `groupID`, in the case of secure multi-party data collaboration on trusted hardware. The user's credentials are checked against a local datastore. While we did not integrate OAuth 2.0 [23] for this iteration of Lens, we adopted a similar paradigm for our authentication/authorization: upon authenticating with the server, the user receives an access token, which the user passes in along with each subsequent request. This access token was designed to have authorization scopes built in.

4.2.2 Uploading Data

When the client uploads personal data through the CLI, the frontend performs a recursive zip on the target directory and makes a HTTPS request to the server with the zip file included in the request body. After authenticating the user via the access token, the server unzips and stores the file in part of trusted memory. The server file system contains */temp* and */cache* directories, which are periodically cleaned up to make room for incoming files and storage needed by the modules.

4.2.3 Module Integration

Core third-party module logic is broken into two stages: the preprocess stage, and the query stage.

Preprocess Stage (e.g. Model Training): While the preprocessing logic is completely offloaded to third parties developing compute modules, the server validates the request body, according to the requirements supplied by the third party through a `PreprocessRequirements()` function. The server then executes the preprocessing function in the trusted execution environment and relays the response back to the user through the encrypted TLS channel.

Query Stage (e.g. Model Inference): Just like preprocessing, the querying logic is completely offloaded to third parties developing compute modules. After validating the request body, according to the requirements supplied by the third party through a `QueryRequirements()` function, the server executes the query on the preprocessed data in the trusted execution environment and relays the response back to the user through the encrypted TLS channel.

5 Modules

5.1 Developing Modules

One of the main contributions of Lens is the ability for service providers to develop and expose compute modules for users to generate rich insights on their data. The steps to developing a Lens module is very straightforward. Code Listing 1 details the module interface, which lists the required public functions for a module to work with Lens. In particular, a module must relay the required inputs for preprocessing and querying the data. Code logic for personal data ingestion and analysis are reserved for the `Preprocess()` and `Query()` functions, which can be called by the command-line interface.

```
type ModuleInterface interface {
    NeedsPreprocessing() bool

    PreprocessRequirements() (requirements []string)
    Preprocess(userId string, parameters interface{}) (response interface{}, err error)

    QueryRequirements() (requirements []string)
    Query(userId string, route string, parameters interface{}) (response interface{}, err error)
}
```

Listing 1: Module Go Interface

5.2 Encrypted Search over Facebook Messenger

A particularly relevant and versatile use case is a search module, which takes in unstructured data and allows for string matching queries on the data. A search module can be used as the building block for more complex applications or provide standalone value.

This module takes a user's Facebook data download as a required dataset.

During the `Preprocess()` function routine, this module builds a search index on the inputted personal data and stores in the index in trusted memory, ensuring security at rest. Though the search index works well on any unstructured data, we applied the search module to Facebook Messenger chat data, downloading a copy of our Messenger chat data and performing search queries over the data.

Once the search index has been built, queries on the search index are extremely lightweight, as a search query simply aggregates matches against the search index and returns the matching messages. The search engine exposes a single `lookup` route with a `keyword` parameter to the user-facing `Query` function.

5.3 Health Metrics

The other module we use to demonstrate the efficacy of multi-party data interaction and computation with Lens is a health metrics application. With the prominence of smartphones and wearable health-tracking devices [34], copious data on users' daily movement and activity are tracked and recorded. This module takes in the daily health summary of multiple users from Apple Health and offers support for multiple queryable insights, both surrounding individual health statistics and group statistics. Of note, this trivial proof-of-concept module provides support for querying which individual was the most active (completed the most steps) during a given month.

To provide support for multi-party data collaboration, Lens associates users with a `groupID` and allows anyone with the correct `groupID` to make queries against the group dataset. Thus, users who would like to compare health in a privacy-preserving manner would be associated with the same `groupID`, allowing them to preprocess and query such data individually.

6 Evaluation

In this section, we demonstrate that Lens represents comparable performance to baseline data analysis, with the added benefit of confidentiality and integrity on user data and module application binaries, quantify its overhead in comparison to an insecure baseline, and measure the performance of Lens against various forms and distributions of data.

6.1 Experimental Setup

Our experiments were run on a single Azure DC1sv3 machine with SGX hardware on an Intel Xeon Platinum 8370C CPU @ 2.80GHz. Our computer was provisioned with with 8 GiB of RAM and 30 GiB Standard SSD LRS storage.

6.2 Experimental Results/Metrics of Success

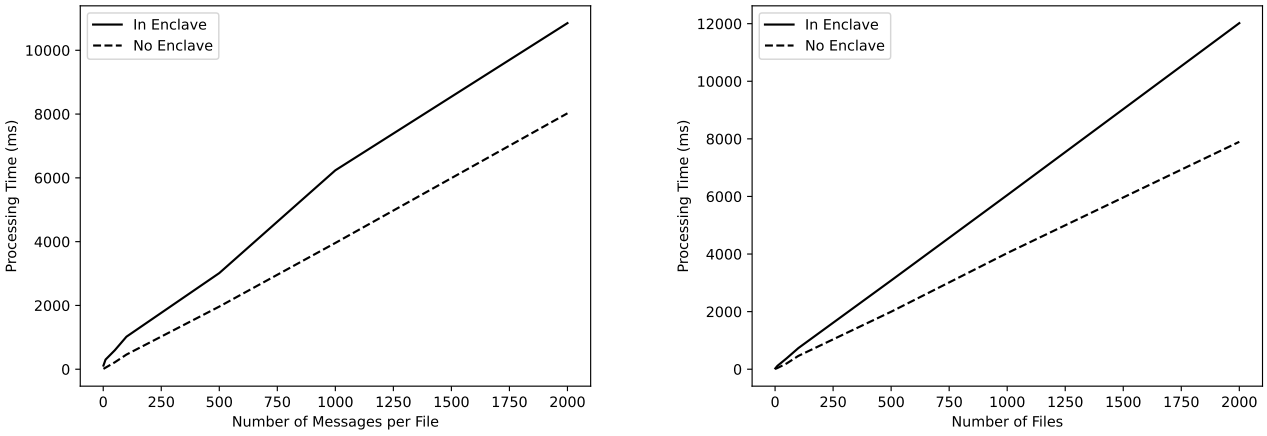
For our evaluation, we measured latency of the search index application. We chose this application as our primary indicator of performance due to the non-trivial computation required to process hundreds of thousands of messages during the index-building stage, the memory requirements of an in-memory search index, and the practicality of search as a semi-realistic application on the Lens framework.

We load-tested the search index application with two scaling metrics:

1. Fixed number of chats per file, scaling number of files (Figure 2a).

2. Scaling number of chats per file, fixed number of files (Figure 2b).

We performed this testing both **inside of the enclave**, and **outside of the enclave**. As implemented, the cost of running the code in-enclave (vs. out-of-enclave as an insecure baseline), is a 52% slowdown when scaling the number of files and a 34% slowdown when scaling the number of chats per file. Since system calls are the dominant source of slowdown



(a) Fixed number of files, scaling number of messages per file

(b) Fixed number of chats per file, scaling number of files

Figure 2: Performance for Processing Meta Messenger Conversations

when working in SGX enclaves, it makes sense that the file scaling has the worse scaling behavior as the file contents remain fairly small for all of the message sizes we examined.

Note that because EGo does not support the separation of trusted and untrusted components, it is likely that these numbers could both be substantially improved by carefully separating out the two components. Since EGo faithfully translates system calls without any batching and without knowledge of the underlying workload, it is likely that building a system to batch inputs and manage working memory carefully could significantly reduce the slowdown, which is already orders of magnitude less than slowdowns observed for cryptography-based solutions. These numbers are in-keeping with general purpose solutions designed to run software as-is in enclaves, so we are optimistic that future work can reduce costs somewhat below these general-case numbers by exploiting compile-time knowledge of program behavior.

6.2.1 Note to the Reader

These evaluations are, as a result, based on high-level performance metrics. The primary insight we derive from this evaluation is the type and order of magnitude of performance hit incurred by this approach to privacy preserving computation. Confirming that the slowdowns we suffer are limited helps illustrate that the user experience provided by Lens would be not at all unpleasant so long as the underlying queries are themselves relatively low-latency (which they are).

This evaluation would significant benefit from finer-grained measurement across all execution and a wider variety of sample datasets and queries — we only had access to the researchers' personal datasets (and synthetically generated datasets based on observed patterns) and little insight into what typical queries might look like. Furthermore, since we did not solicit other developers to prepare applications for our platform, it would be interesting to investigate how performance scales for applications developed with less insight into Lens' design.

7 Discussion/Future Work

7.1 Application Porting

Due to time constraints, we were unable to assess and demonstrate the feasibility of migrating existing Golang applications onto the Lens framework. This would be an immediate next step, specifically to identify the points at which Ego/OpenEnclave limit "lift-and-shift" application logic. We would also like to further explore the feasibility of writing self-contained applications that don't rely on outbound network requests and/or multiple services - alternatively, exploring extensions to Ego/OpenEnclave that support these features would be viable future studies as well.

7.2 Disk Sealing

Ego allows application developers to define custom mount points that apply to the file system present in the enclave. These mounts may be of type `hostfs`, where the mount point is on the (untrusted) host file system itself, or `memfs`, where the mount point is to a (trusted) in-memory filesystem. For our framework, we implemented an application-layer `hostfs` file system interface - a package to open/read/write/delete contents to disk, encrypted and authenticated using the sealing keys provided by OpenEnclave. However, because this sealed file system is at the application level, our current approach

doesn't support third-party dependencies that rely on on-disk storage. One area of research could be to build this sealing layer directly into the Ego framework (e.g. migrate all I/O calls at compile-time over to sealed I/O calls).

7.3 Real-Time Support

The proof-of-concept implementation of Lens is limited to user-provided datasets (e.g. GDPR Data Downloads). In real-world implementations, third-party applications often require real-time data sources - emails, locations, photos, and messages

For real-time support, there must be a method to manage a user's third-party credentials while preventing any individual from seeing this data. For this to happen, more consideration would need to be given to the network I/O allowed by the enclave; for example, allowing outbound network requests (e.g. a query to fetch some updated email data from Google servers) would potentially open user data to side channel attacks, where an untrusted codebase could exfiltrate user data through a network request to a Google server. This attack could, though, be mitigated by wrapping all network I/O in trusted Lens code - where Lens code handles all data downloads/uploads, and third-party modules are blocked from all network I/O, such as in the current architecture - but would increase the size of the TCB as a tradeoff.

7.4 Confidential Service Mesh

In order to support scalability in a real-world implementation, the Lens architecture would have to be ported over to a confidential Kubernetes service mesh, where each component (e.g. database, web server, each application module) could scale independently of all others. Future research could include implementing such a system, with pairwise remote attestation, large-scale data storage (instead of on-VM disk storage), and query synchronization.

7.5 Mutually Untrusting Services

Our original goal was to build this system on top of Ryoan [24], a system that supports unstrusted computation on secret data. Future studies could explore migrating our current architecture (where each module is built directly into the core framework) to a model that functions more like the iOS App Store:

- Services are built using well-defined API's provided by Lens.
- Developers submit services for review by the Lens team. This review would run automated testing over the module, and not necessarily require knowledge of the module's internal secrets.
- Approved services are made available to users, and run on the Lens backend. Because services are running on the Lens backend, they run in a sandboxed container and are **untrusted** by both the user AND the Lens framework.
- The Lens framework is not able to view any secrets contained within the third-party service; it's executed directly within an isolated sandbox inside the enclave.

7.6 Side Channel Attacks & Access Pattern Protection

Lens does not currently provide mitigations for side-channel attacks on SGX. Many side-channel attacks known in SGX could be fixed via hardware changes, microcode changes, or software changes, but this is laborious and did not fit into our time window. Another major leakage vector is side-channels — ORAM is a traditional solution to completely hiding these, but while usable for some purposes, the current state of the art is still too slow for our user-facing applications.

An alternative to traditional obliviousness-based approaches to access pattern hiding is Pancake [22], which uses *statistical* access pattern hiding. It simply pads access patterns up to approximate uniformity, resulting in an adversary being unable to extract information from the access pattern traces other than potential timing information, which can be masked by noise insertion. We attempted to integrate Pancake into this work, but its codebase is unwieldy and does not lend itself well to enclave adaptation — this is a ripe area for future work.

8 Conclusion

In this paper, we proposed Lens, a service-oriented platform for privacy-preserving computation over personal data. Lens contributes an end-to-end workflow for users to generate insights on personal data using third-party compute modules, while enabling relevant parties to query these insights in a privacy-preserving manner. Lens opens the door to the liberal exchange of data between users and other parties without requiring trust from either party to create a symbiotic relationship.

References

[1] Cli. https://github.com/urfave/cli.

[2] Cloud Computing Services. https://cloud.google.com/.

[3] Cloud Computing Services | Microsoft Azure. https://azure.microsoft.com/en-us/.

[4] Cloud Services - Amazon Web Services (AWS). https://aws.amazon.com/.

[5] Ego: Build confidential go apps with ease. https://www.ego.dev/.

[6] Open Enclave SDK. https://openenclave.io/sdk/.

[7] ABADI, M., CHU, A., GOODFELLOW, I., MCMAHAN, H. B., MIRONOV, I., TALWAR, K., AND ZHANG, L. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), pp. 308–318.

[8] ABOWD, J. M. The us census bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018), pp. 2867–2867.

[9] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., ET AL. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 689–703.

[10] BRAKERSKI, Z., GENTRY, C., AND VAIKUNTANATHAN, V. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT) 6*, 3 (2014), 1–36.

[11] BRAKERSKI, Z., AND VAIKUNTANATHAN, V. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing 43*, 2 (2014), 831–871.

[12] BUFALIERI, L., LA MORGIA, M., MEI, A., AND STEFA, J. Gdpr: when the right to access personal data becomes a threat. In *2020 IEEE International Conference on Web Services (ICWS)* (2020), IEEE, pp. 75–83.

[13] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 253–264.

[14] CORP, I. Product change notification 114074-00.

[15] DE HERT, P., PAPAKONSTANTINOU, V., MALGIERI, G., BESLAY, L., AND SANCHEZ, I. The right to data portability in the gdpr: Towards user-centric interoperability of digital services. *Computer law & security review 34*, 2 (2018), 193–203.

[16] DWORK, C. Differential privacy. In *Automata, Languages and Programming* (Berlin, Heidelberg, 2006), M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds., Springer Berlin Heidelberg, pp. 1–12.

[17] DWORK, C. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation* (2008), Springer, pp. 1–19.

[18] DWORK, C., ROTH, A., ET AL. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci. 9*, 3-4 (2014), 211–407.

[19] GENTRY, C. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[20] GENTRY, C., HALEVI, S., AND SMART, N. P. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2012), Springer, pp. 465–482.

[21] GRUBBS, P., KHANDELWAL, A., LACHARITÉ, M.-S., BROWN, L., LI, L., AGARWAL, R., AND RISTENPART, T. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2451–2468.

[22] GRUBBS, P., KHANDELWAL, A., LACHARITÉ, M.-S., BROWN, L., LI, L., AGARWAL, R., AND RISTENPART, T. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2451–2468.

[23] HARDT, D., ET AL. The oauth 2.0 authorization framework, 2012.

[24] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 533–549.

[25] KAPLAN, D., POWELL, J., AND WOLLER, T. Amd memory encryption. *White paper* (2016).

[26] KNAUTH, T., STEINER, M., CHAKRABARTI, S., LEI, L., XING, C., AND VIJ, M. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).

[27] MANYIKA, J., CHUI, M., BROWN, B., BUGHIN, J., DOBBS, R., ROXBURGH, C., HUNG BYERS, A., ET AL. *Big data: The next frontier for innovation, competition, and productivity*. McKinsey Global Institute, 2011.

[28] MARTINS, P., SOUSA, L., AND MARIANO, A. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv. 50*, 6 (Dec. 2017).

[29] MCSHERRY, F. D. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), pp. 19–30.

[30] MEHMOOD, A., NATGUNANATHAN, I., XIANG, Y., HUA, G., AND GUO, S. Protection of big data privacy. *IEEE Access 4* (2016), 1821–1834.

[31] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 279–296.

[32] NAEHRIG, M., LAUTER, K., AND VAIKUNTANATHAN, V. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), pp. 113–124.

[33] POPA, R. A., REDFIELD, C. M., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[34] RINGEVAL, M., WAGNER, G., DENFORD, J., PARÉ, G., AND KITSIOU, S. Fitbit-based interventions for healthy lifestyle outcomes: systematic review and meta-analysis. *Journal of medical Internet research 22*, 10 (2020), e23954.

[35] STEHLÉ, D., AND STEINFELD, R. Faster fully homomorphic encryption. In *International Conference on the Theory and Application of Cryptology and Information Security* (2010), Springer, pp. 377–394.

[36] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 283–298.

[37] ZISSIS, D., AND LEKKAS, D. Addressing cloud computing security issues. *Future Generation computer systems 28*, 3 (2012), 583–592.