Marble: Making Fully Homomorphic Encryption Accessible to All

Alexander Viand
Department of Computer Science
ETH Zurich, Switzerland
viand@inf.ethz.ch

Hossein Shafagh
Department of Computer Science
ETH Zurich, Switzerland
shafagh@inf.ethz.ch

ABSTRACT

With the recent explosion of data breaches and data misuse cases, there is more demand than ever for secure system designs that fundamentally tackle today's data trust models. One promising alternative to today's trust model is true end-to-end encryption without however compromising user experience nor data utility. Fully homomorphic encryption (FHE) provides a powerful tool in empowering users with more control over their data, while still benefiting from computing services of remote services, though without trusting them with plaintext data. However, due to the complexity of fully homomorphic encryption, it has remained reserved exclusively for a small group of domain experts. With our system Marble, we make FHE accessible to the broader community of researchers and developers. Marble takes away the complexity of setup and configuration associated with FHE schemes. It provides a familiar programming environment. Marble allows rapid feasibility assessment and development of FHE-based applications. More importantly, Marble benchmarks the overall performance of an FHE-based application, as part of the feasibility assessment. With real-world application case-studies, we show the practicality of Marble.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**; **Management and querying of encrypted data**;

KEYWORDS

Encrypted Applications; Homomorphic Encryption Schemes

ACM Reference Format:
Alexander Viand and Hossein Shafagh. 2018. Marble: Making Fully Homomorphic Encryption Accessible to All. In *6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3267973.3267978

1 INTRODUCTION

Maintaining user privacy in the era of cloud-computing is a complex challenge. Users demand increasingly powerful applications that require outsourcing storage and computation from client devices to powerful servers in the cloud. However, outsourcing puts user data at risk and threatens privacy. This can be resolved by encrypting user data on the client side with efficient symmetric key-based schemes, e.g., AES-CTR, before sending it to the cloud. However, this leaves the abundant computation resources on the cloud unused and introduces significant application latencies, as the user has to download large segments of the ever growing datasets locally, to perform computations on top, e.g., data analytics. By leveraging *fully homomorphic encryption* (FHE), the user can outsource arbitrary computations on their encrypted data to a remote server without revealing the plaintext data nor trusting the cloud with encryption keys. With FHE the server learns neither the inputs, outputs, nor intermediate results of the computation.

FHE is a young research field and poses a high barrier of entry, specifically for non-cryptographers. The broader research community and developers make the common assumption that real-world FHE-based applications are either infeasible or impractical due to a significant slowdown of several orders of magnitude. For instance, that operations taking nanoseconds on a modern CPU operating on plaintext data can take several seconds to minutes on FHE-encrypted data. It is however less known that despite this slowdown, there are scenarios where FHE can be practical or near-practical. For instance, Wang et al. show a system to privately outsource feature-extraction to the cloud using FHE [40]. Their solution outperforms local computation for reasonably sized images, even with the overhead of FHE. Çetin et al. present an auto-completion system that uses FHE [16] and achieves latencies of roughly one second.

Estimating the overhead required for FHE in a given application is hard for developers outside of the cryptography research community. Often, it is not apparent whether or not an application will be practical until an FHE-based implementation is attempted. This can be a daunting task requiring a significant time investment which discourages developers and researchers from exploring FHE. For instance, consider the case of a researcher in the field of biometrics-based authentication who is exploring alternative secure designs that allow for reducing the trust in the storage- and computation-providers (i.e., cloud) for a biometrics-based authentication system. It is desirable for her to understand the overhead of a FHE-based version of her authentication algorithm without necessarily being a crypto expert.

We therefore propose Marble, a high-level framework that allows developers without a background in FHE or cryptography to easily prototype FHE-based applications. Marble handles parameter choices, cryptosystem set-up, and behind-the-scenes emulation of binary arithmetic circuits. The framework can automatically analyze and benchmark applications to report the FHE overhead.

WAHC '18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5987-0/18/10.
https://doi.org/10.1145/3267973.3267978

Our goal is to make exploring FHE more accessible, enabling developers to quickly discover the applicability of FHE to their scenario. The performance of the underlying FHE scheme is an inherent key factor in Marble. Hence, Marble's modular design makes the currently most promising schemes accessible. To facilitate this, the underlying cryptosystem implementation can easily be replaced as advances in FHE research are made.

To realize our system, we address the following challenges: There are two common approaches to computing with FHE; direct evaluation of polynomial functions and binary circuit emulation for more general functions. *(i)* Choosing either of these approaches for an application is non-obvious and changing it during development can lead to large amounts of wasted effort. *(ii)* Different FHE implementations have different setup, initialization, and parameter selection requirements. Depending on the library in question, a significant amount of 'boiler-plate' code and implementation-specific insight is required to compile, initialize, and use them. *(iii)* Evaluating FHE-based applications can be difficult since runtime varies greatly between different machines and different FHE implementations. Instead, a better measure for an application's performance is to quantify its inherent complexity rather than merely runtime.

The main contribution of this work is an easy-to-use framework that enables developers to write FHE-based programs as if they were plaintext programs, automatically selects appropriate parameters, handles initialization and setup, and evaluates programs both in terms of runtime as well in terms of machine- and implementation-independent characteristics. We demonstrate the usability of our framework through the implementation and evaluation of several example applications. We calculate the hamming distance between two encrypted vectors, implement encrypted location-based ads, and show how to use FHE for privacy-preserving face recognition. We make the source-code of our prototype implementation and a ready-to-use docker image publicly available[1].

2 BACKGROUND

Understanding the overall capabilities and limitations of FHE is important not only because they motivate the design choices we make in Marble, but also because designing FHE-based applications requires a mindset that slightly differs from standard programming to accommodate for these limitations. We briefly introduce a high-level notion of fully homomorphic encryption, focusing on aspects that influence the design of FHE-based applications. Afterwards, we discuss how to leverage FHE for computations.

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) allows a third party to perform computations on encrypted data, learning neither the inputs nor results of the computation. The concept was first proposed by Rivest et al. in the 1970's, shortly after the development of public-key encryption [32]. However, it remained an unsolved problem until 2009, when Craig Gentry presented a breakthrough result [19].

A *homomorphic* encryption scheme is an encryption scheme where there exists a homomorphism between operations on the plaintext and operations on the ciphertext. In other words: If we have some operation (e.g., +) on the plaintexts then there is an operation $\oplus$ on the ciphertexts so that $\mathrm{Dec}(\mathrm{Enc}(x+y)) = \mathrm{Dec}(\mathrm{Enc}(x) \oplus \mathrm{Enc}(y))$. A *fully* homomorphic encryption scheme is one that supports an arbitrary mix of additions and multiplications[2].

There are a variety of different FHE constructions that differ in their hardness assumptions and approaches [11]. However, the most important aspects for designing FHE-based applications are either inherent in FHE or shared among all current schemes. For example, they generally support the plaintext space $\mathbb{Z}_p$, i.e., the integers modulo $p \in \mathbb{N}$, for a parameter p.

All current FHE schemes also have the property that ciphertexts *degrade* during computation. This is because they encrypt a value m as a ciphertext $c = m' + \epsilon$ for some small *noise* ϵ and a suitable encoding m' of the plaintext. When this noise grows too large, decryption will no longer return the correct plaintext. Different schemes adopt different noise-management techniques [4], with a maximum *noise threshold* that derives from the parameters of the scheme. Generally speaking, a larger noise threshold results in slower homomorphic operations. For all schemes, additions increase the noise much less than multiplications. Especially chained multiplications, i.e., multiplications of already multiplied ciphertexts cause an explosive noise growth. This is why additions are computationally more efficient than multiplications.

2.2 Computing with FHE

With additions and multiplications, we can compute any polynomial function over the encrypted data. This class includes functions like the squared ℓ_2-norm and gradient descent. However, some important functions used in programming are not polynomial. Most importantly, comparisons (e.g., ordering and equality) are non-polynomial and they are essential to many computations.

To move beyond polynomial functions to arbitrary functions, one can use an FHE scheme on the plaintext space $\mathbb{Z}_2$, i.e., integers modulo 2. In this binary domain, multiplication emulates a logical `and`-gate while addition emulates an `xor`-gate. From these two gates, any binary circuit can be constructed. Together with memory, binary circuits are Turing-complete, i.e., they can emulate any arbitrary computation. However, the overhead of this emulation is dramatically large. Instead, we need to consider what we can express without requiring an impractical emulation. We discuss in Section 4 how Marble tackles this issue with a pre-analysis of the plaintext computation.

Compared to a generic program, a circuit has a few key limitations. First, its inputs and outputs are always fixed in size. This can be ignored when computing mathematical functions, but is a challenge for more generic applications. For example, a search application must always return the same number of items, no matter how many results there were. Secondly, the computation must be data-independent. Therefore, we cannot use branching instructions that depend on data, e.g. `if`-statements or loops with dynamic length. We can emulate the result of `if`-statements, however both branches must be evaluated.

[1] Marble is available at https://github.com/MarbleHE

[2] There are some additional formal constraints to exclude trivial constructions, see [20]

3 DESIGN

The barrier of entry for researchers and developers who want to explore FHE is currently very high. Deciding on the scheme and library to use, the appropriate encoding of plaintext values and the parameters with which to instantiate the scheme add a significant amount of up-front complexity. In addition, there is the general overhead of setting up various libraries, learning their large interfaces and setting up a development system. Afterwards, depending on the choice of library and encoding, developers might first have to re-implement common binary arithmetics before they can start to implement their actual application. While these challenges are certainly not insurmountable, they discourage researchers and developers from exploring FHE for their problem space.

With Marble, we lower this barrier of entry and streamline the process of exploring FHE. To achieve this design goal, user-friendliness and ease-of-use are paramount. Moreover, it is also crucial that the expressiveness and power of FHE are not sacrificed in the pursuit of ease-of-use. Hence, we must abstract away different encodings, libraries, and underlying FHE schemes into one unified interface that is expressive enough to allow developers to write efficient applications while being easy to use. This necessitates the ability to automatically select appropriate parameters, instantiate the underlying schemes and libraries, and benchmark the user-provided application.

3.1 Threat Model

When working with FHE, we generally assume an *honest-but-curious* setting, where the adversary follows the protocol correctly, but is interested to learn as much as possible about the data. This is a legitimate trust model, as violations can be detected and lead to financial penalization. Hence, our threat model protects users data from a passive adversary who is interested to learn about data and from data breaches as a consequence of system compromise. Note that our system does not provide assurances about the availability of the system nor the correctness of the final result. Also access patterns of users can be leveraged to leak sensitive information.

3.2 Ensuring Cross-Platform Ease-of-Use

To lower the barrier of entry, one important aspect is providing a simple unified interface for a variety of schemes and libraries. One way to realize this, is creating a domain specific language (DSL) that developers can use to specify their applications. Such a DSL representation is read by a special parser and transpiled into code in a general programming language. While this is practical for scenarios where the standard operations are not conveniently expressed in general programming languages, it also requires developers to learn the syntax and semantic of a new language. In addition, since such a DSL is often a niche language, toolchain support (such as syntax highlighting or debugging) will be limited compared to mainstream languages. Therefore, we instead decided to implement Marble as a C++ library, so that developers can use the development environments that they are familiar with and benefit from the maturity of the C++ ecosystem.

With a C++ based system, however, we cannot simply provide an application package for the main operating systems. Instead developers need to download and install Marble and the FHE libraries it depends on manually. This increases the installation effort which runs counter to our goals. Hence, we provide Marble as a ready-to-run Docker image, where all the necessary libraries and tools are pre-installed and configured. With this, developers can either directly use their favorite terminal-based development environment or IDE with remote debugging support. The current Marble image comes pre-configured to work with Visual Studio and CLion remote debugging and supports most other IDEs. Since Docker is available on all major platforms, it solves the cross-platform issue and due to the lightweight container nature, there is essentially no performance impact on the runtime of the FHE computations [18].

3.3 Choosing an FHE Library

There are a large number of FHE schemes and implementations that differ in a range of aspects. They differ in the underlying hardness-assumptions, their efficiency with regard to computation time or ciphertext size, their support for advanced features like bootstrapping, their plaintext spaces, their support for optimization techniques, and many other aspects. Supporting all FHE implementations would be neither feasible nor sensible, as some implementations strictly outperform their predecessors. Therefore, we must carefully choose which library to use as the cryptographic back-end for our framework.

A recent analysis and comparison of the main branches of current cryptographic research about FHE highlights two promising families of schemes [11]. The first is based on the Brakerski-Gentry-Vaikuntanathan (BGV) construction, whereas the other is based on the Fan-Vercauteren (FV) construction. These two constructions offer different trade-offs with neither strictly outperforming the other. We employ both a BGV-based as well as an FV-based FHE implementation in the back-end of our framework. For BGV, the most mature and stable implementation is by far HElib [23–25] by Halevi and Shoup. On the side of FV, we consider the Simple Encrypted Arithmetic Library (SEAL) [9] to be the most suitable option. With these choices, our framework supports the two most actively developed libraries from two of the most promising approaches to FHE. Thanks to internal abstractions (described in more detail in Section 6), our framework can also easily accommodate other back-end libraries should a new library offer significant performance benefits.

3.4 Supporting Fractional Numbers

Current FHE schemes generally use plaintext spaces based on integers. However, a large number of problems are defined over real numbers. When computing with plaintext data, floating point numbers are used to approximate real numbers. In theory, one could emulate binary circuits for floating point numbers using FHE. However, this is impractically slow because floating-point operations require complex arithmetic and logic circuits. Therefore, we need to represent numbers in a way that is more efficient.

Fixed-point representation is similar to floating point representation. Numbers $x \in \mathbb{R}$ are approximated as $s \approx z \cdot 2^s$ for $z, s \in \mathbb{Z}$. In floating-point representation, every number has a dynamic scaling factor s that is stored together with the mantissa z. In fixed-point representation, the scaling factor s does not depend on the value of the number. This simplifies their implementation considerably,

making them attractive for systems that do not have native floating point instructions like small embedded systems or FHE.

3.5 Translating to low-level FHE Code

To ease the exploration of different applications, we provide a unified high-level abstraction. Developers are able to write their application without having to consider the underlying schemes and encodings. The framework picks a suitable scheme and encoding and allows the developer to easily benchmark the application. Programs look as similar to their plaintext counterparts as possible when considering the semantics of FHE, as elaborated in Section 4. Therefore, instead of exposing details like ciphertexts and encodings, Marble provides a single new type that can be used like a built-in numeric type. In Section 4, we show in detail how developers can write code that is similar to operations on built-in plaintext types, with the exception of encryption and decryption operations.

To achieve this high level of abstraction, Marble internally selects *(i)* the appropriate library, *(ii)* the appropriate encoding, *(iii)* suitable parameters, and *(iv)* translates the plaintext operations into the corresponding FHE operations. In the initial step, Marble simulates the entire computation, keeping track of the (bit-)sizes of plaintext values, their scale (if in fixed-point representation), and the maximally required multiplicative depth. From these results, it then decides whether to use e.g. bit-slicing or direct encoding based on whether or not any non-polynomial operations were performed. Marble employes simple heuristics to decide whether or not to use bootstrapping, which library to use and which encoding to use for the plaintext values, if not using bit-slicing. These heuristics are not meant to be optimal, but represent a solid starting point for exploring FHE. As the underlying schemes become more efficient, these heuristics can be seamlessly updated to provide better performance.

4 USING MARBLE

We now highlight how Marble can be leveraged to quickly go from an idea to a working FHE application and benchmark it. We introduce the most important features of the framework with the help of three example applications: We start with a simple calculation of the hamming distance between two encrypted vectors. Afterwards, we look at two considerably more complex examples: location-based search and face recognition. These showcase the capabilities and limitations of Marble and FHE in general. These examples are not intended to be full solutions to the problems they consider. Instead, these examples are meant to illustrate the speed with which one can explore the applicability of FHE to a certain domain. Where appropriate, we highlight what additional considerations are required to turn these into full solutions.

Our system abstracts away the underlying implementation details of FHE and provides a plaintext-like interface. However, it cannot abstract away the semantic differences between working with FHE encrypted values and plaintexts. Therefore, we also use these examples to highlight these differences. In addition, we show general strategies on how to reformulate problems to make them suitable for FHE and how to use SIMD-style operations to greatly improve the performance.

4.1 Hamming Distance

For our first example we look at privately computing the hamming distance between two encrypted binary vectors. The hamming distance counts the number of positions where the two vectors differ. For two vectors $v = (v_1, v_2, \ldots, v_n)$ and $u = (u_1, u_2, \ldots, u_n)$ the hamming distance is the number of positions where $v_i \neq u_i$. The Hamming Distance is used, for example, in biometric access control but is also a common building block in many other domains.

A simple algorithm to calculate the hamming distance over plaintext data might look as follows (we ignore size differences between the two vectors for clarity of presentation):

```
int hd_plaintext(vector<bool> v, vector<bool> u) {
    int sum = 0;
    for(int i = 0; i < v.size(); ++i) {
        sum += (v[i] != u[i]);
   }
   return sum;
}
```

We can write this algorithm in a nearly identical way using our framework:

```
void hd_enc(vector<M> v, vector<M> u) {
    M sum = 0;
    for(int i = 0; i < v.size(); ++i) {
        sum += (v[i] != u[i]);
    }
    output(sum);
}
```

Where `M` is the unified interface type defined by Marble. Apart from changing the types, little has changed in the program. One minor adjustment that is necessary due to language limits in C++ is that our function now returns `void`, i.e., has no return value. Instead, Marble provides a replacement in the form of the function `output`. In order to analyze and benchmark this example, we simply pass the function to the framework:

```
int main() {
    vector<bool> v = {0,1,1,0,0/*...*/);
    vector<bool> u = {1,0,1,0,1/*...*/);

    vector<M> v_enc = encrypt(v);
    vector<M> u_enc = encrypt(u);

    // Simulates the execution and
    // reports e.g. multiplicative depth
    M::analyse(bind(hd_enc,v_enc,u_enc));

    // Benchmarks the application,
    // using the most appropriate settings
    M::evaluate(bind(hd_enc,v_enc,u_enc));
}
```

The code first encrypts two vectors, then calls into Marble to generate a report on the computation complexity (`analyse`) and to benchmark the FHE computation (`evaluate`). The `bind` function is part of the C++ standard library and is required to make a function and its arguments easily passable to another function.

The results of both calls can be found in Section 5, where we discuss the performance of these examples in more detail.

While this implementation was close to the plaintext source code, it can easily be made significantly faster with a small amount of effort. FHE tends to lend itself naturally to Single Instruction, Multiple Data (SIMD) style parallelism [38], this is commonly referred to as *batching*. With batching, one can pack multiple values into a single ciphertext. We can think of a ciphertext as a vector, with each *slot* holding a value. Homomorphic operations then act like SIMD operations, where an operation is applied to each slot in the ciphertext at the same time. Combined with the ability to rotate values between slots, this becomes a powerful tool for optimization. Marble supports batching with only small changes to the code:

```
void hd_batched(M v, M u) {
   M diff = (v != u);
   diff.fold(sum);
   output(diff);
}
```

Where the only change to the `main` function is that we now have `M v_enc = encrypt(v, /*batching = */ true);` and the same for u, i.e., we indicate that we want batching from the encryption and get only a single encrypted value rather than a vector of encrypted values. In the batched function, we have replaced the for loop with a single SIMD inequality operation and the summation is done using the `fold` function that combines elements from all slots according to a function specifier, here `sum`. Other supported operations include `min`, `max` or generally any function that takes two `M` values as input and returns one `M` value. This optimization is particular to the inherent benefits of SIMD to alleviate the otherwise expensive FHE computations. This is why Marble clarifies this fundamental difference with examples.

4.2 Location-based Advertising

We next consider the example of location-based advertising, i.e., enhancing the relevance of ads, for example in mobile browsing, by factoring in the user's location. Since location data is inherently privacy sensitive, we explore how to achieve this without leaking the user's location data. In online ads, the ad provider has a pool of possible ads and is interested to display the ad that is most relevant to maximize the chance that a user clicks on the ad. When considering location-based advertising, the likelihood of a user clicking on an ad for e.g., a cinema in a different state are generally lower than for a location that is nearby. For a given ad in a pool of possible location-based ads to show, we model this probability, commonly referred to as the click-through rate, for ad i as $CTR_i = r_i * \frac{1}{distance(user,\ place_i)}$ where r_i models the relevance of the ad to the content of the website. For the locations, we consider a two-dimensional grid with coordinates x and y. We use FHE to compute the most relevant ad without revealing the location of the user to the ad provider. Given the homomorphically computed index of an ad, one could use private information retrieval (PIR) techniques to allow the client application to retrieve the ad itself without leaking information to the server.

Alternative Approaches. Note that various alternative approaches to locational privacy exists with varying threat models and guarantees, however for this paper we are naturally interested in applying FHE to the problem. Most existing locational privacy systems work by masking the client's true position in a set of possible positions (either a closed region or a variety of points all over the place). This means that the server returns results not only for the true position but also for all other positions in the set, with the client discarding the non-relevant ones.

FHE Implementation. Our task is to implement a function that takes an encrypted user location $(u_x,\ u_y)$ and a list of relevances r_i's and locations $place_i = (x_i,\ y_i)$ (available to the server in plaintext) and return the ad with the best click-through rate $CTR_i = r_i * \frac{1}{distance(user,\ place_i)} = r_i * \frac{1}{\sqrt{(u_x - x_i)^2 + (u_y - y_i)^2}}$, where we used the formula for the euclidean distance between two points.

Note that there are several optimizations we can make to the formula for CTR_i that do not change the ordering of the advertisements. Since both r_i and the distance are always positive, squaring the whole term changes the maximum value, but not which element is the maximum. That way we can eliminate the expensive square root operation in the euclidean distance calculation. Similarly, we see that the ad with maximum CTR is the same as the ad with minimum $\frac{1}{CTR_i}$, i.e., we can invert the whole term. We also round the inverted $\frac{1}{r_i}$ to integers for efficiency. This is helpful since r_i is a plaintext value and can be inverted effectively "for free", whereas a homomorphic division is incredibly expensive computationally.

With these simplifications, we need to implement a function that takes a list of relevance factors r_i, a list of place coordinates x_i, y_i (i.e., the location of the store that is being advertised) and an encryption of the position of the user (x_u, y_u) and returns i (encrypted) so that $\frac{1}{r_i} * ((u_x - x_i)^2 + (u_y - y_i)^2)$ is minimal:

```
void ads(M u_x, M u_y, M r_inv , M x, M y) {
   M d1 = u_x - x;
   M d2 = u_y - y;
   M ctr =  r_inv * ( d1 * d1 + d2 * d2 );
   // SIMD-minimum-index
   M index;
   M min = ctr.fold(min_with_index, index);
   output(index);
}

int main() {
   // Coordinates and relevance of advertised places
   vector<int> x = {114, 254, 35/*...*/};
   vector<int> y = { 76, 112, 40/*...*/};
   vector<int> r_inv = {(1/0.24),(1/0.68)/*...*/};

   // User coordiantes
   vector<int> u_x = {17, 17, 17 /*...*/);
   vector<int> u_y = {158,158,158/*...*)};

   M x_ = encode(x); M y_ = encode(y);
   M rinv_ = encode(r_inv);
   M ux_enc = encrypt(u_x); M uy_enc = encrypt(u_y);

   M::evaluate(bind(ads,ux_enc,uy_enc,rinv_,x_,y_));
}
```

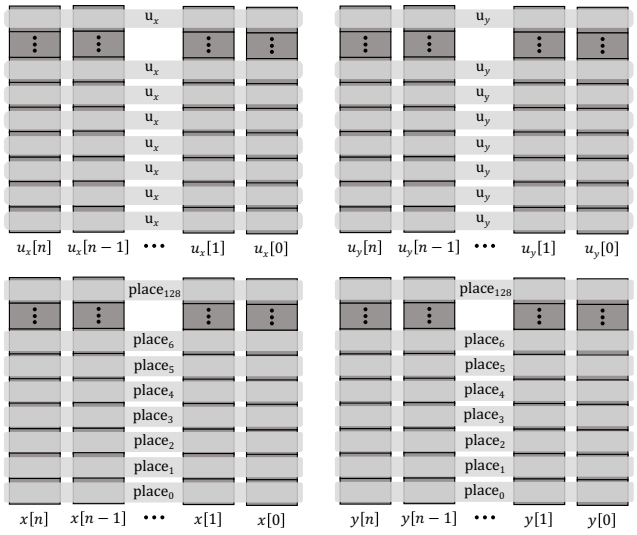


Figure 1: Visualization of the inputs to the function `ads(..)` with the batching scheme highlighted. Each input u_x, u_y (user coordinates) and x, y (place coordinates) is split into n bits, each encrypted into a separate ciphertext. With batching, each ciphertext resembles a vector. For the user coordinates, the same value is repeated in each slot, whereas for the places we encrypt the coordinates of $place_i$ in slot i. This way, a few SIMD operation will compute the distances for all places at the same time.

We again leverage batching to speed up the computation. While our inputs (coordinates of the user and the advertised places) are not yet vectors, we can still employ batching to our advantage: We store the coordinates for the i-th place in the i-th slot of the ciphertexts of `x` and `y` For the user coordinates `u_x` and `u_y` we simply replicate the same value across all slots. Figure 1 illustrates this and also shows that `u_x`, `u_y`, `x` and `y` each corresponds to n ciphertexts internally, one for each bit, as this example requires bit-slicing due to the use of comparisons. This batching scheme allows us to compute the distance between the user and multiple advertised places at the same time using SIMD operations.

In addition to the batching, this example also uses `encode` rather than `encrypt` for the inputs that are known to the server (i.e., not secret). This indicates to the system that these values are available as plaintext. This facilitates several optimization both at the level of Marble and in the underlying FHE libraries and reduces the complexity of the FHE calculations.

4.3 Face Recognition

Face recognition is the task of identifying a face in a picture, given a set of "known" faces. It is by its very nature a privacy-sensitive application. Therefore, there have been many different solutions to achieve privacy-preserving face recognition, mostly based on secure multi-party computation or partially homomorphic encryption [1, 15, 34]. In this section, we present a face recognition system that uses FHE to carry out the matching process on the server.

We consider a scenario where a user has a picture that she wants to perform face recognition on, against a database of (encrypted) face-templates that she has previously stored on the server. This can apply to labeling of private photo albums or a law-enforcement scenario of finding a suspect match within the large encrypted database without disclosing the plaintext images nor the images of the suspects to the storage provider. First, the client device creates a more efficient representation of the image, using for example FaceNet [35]. The FaceNet network is a face recognition system that generates compact representations from input images. It encrypts the resulting vector of 128 8-bit values and uploads it to the server. The server calculates the best match by finding the face-template that has the smallest euclidean distance from the input. Finally, it returns the index of the best match. For the server, this is simply a higher-dimensional version of the previous example.

Alternative Approaches. Ciang, Tang et al. show a protocol for privacy-preserving face recognition using a combination of FHE and symmetric encryption [41]. The setting differs from ours in that there are three parties: The client (image owner), the database owner and a third party cloud server used for outsourcing the computation. The image owner and the database owner should not learn each others inputs. In contrast, our scenario is simpler, with only the client who holds the keys to the current image as well as to the database and then the cloud service that is used for outsourcing. As a consequence, our system has much lower online communication complexity than theirs.

FHE Implementation. Since this problem requires a similar computation from the server, the code is similar to that of the previous example:

```
void faces(M in, M db, int dim, int n) {
   M diff = in - db;
   M sq = diff * diff;

   // SIMD-summation over all dimnensions of each face
   M dist = sq.fold(sum,dim);

   // SIMD-minimum-index
   M index;
   M min = dist.fold(min_with_index, index, dim, n);
   output(index);
}
int main() {
   // Database of faces templates
   vector<vector<int>> db /*= {face1, face2, ...}*/;
   int n = db.size();
   int dim = db[0].size();

   // Input face representation
   vector<int>  in = {76, 112, 40/*...*/};

   // Batched Encryption
   vector<int> db_batched, in_batched;
```

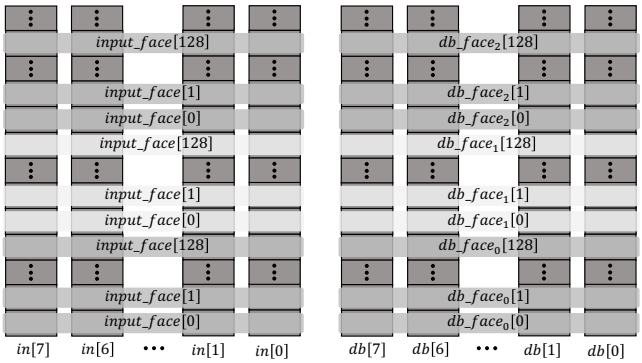


Figure 2: Visualization of the inputs to the function `faces(..)` with the batching scheme highlighted. Each face-representation (*input_face*) or face-template (*db_face*) is a 128-dimensional 8-bit vector. We split both inputs `in` and `db` into a ciphertext per bit. With batching, we can consider each ciphertext as a vector and we use blocks of 128 slots to hold one face-representation or template. While we repeat the same face-representation for the input, the `db` ciphertexts contain a different face-template in each block. This allows SIMD operations to calculate the distance between the input face and all face-templates at the same time.

```
    for(int i = 0; i < n; ++i) {
        db_batched.push_back(db[i]);
        in_batched.push_back(in);
    }
    M db_enc = encrypt(db_batched, /*bitSize=*/8);
    M in_enc = encrypt(in_batched, /*bitSize=*/8);

    M::evaluate(std::bind(faces,in_enc,db_enc,dim,n);
}
```

However, we encrypt the values differently compared to the location-based advertising. Instead of having one SIMD vector for each dimension of all the face-representations in the database or one SIMD vector for all dimensions of each face-representation, we batch values from all dimensions into a single SIMD vector, with each block of 128 elements representing one face. This is illustrated in Figure 2. After calculating the squared differences, we use an overloaded version of `fold` that folds only up to n elements of the SIMD vector together, so that we sum only the dimensions corresponding to a single face. Finally, the minimum index function takes another parameter that indicates that it should ignore all elements except every m-th one. In fact, all of the SIMD-element-manipulating operations in Marble provide overloaded versions that operate only on the specified sub-vectors.

We have seen three applications of increasing complexity. In the next Section, we compare and contrast their performance in order to give an overview for the overheads imposed by FHE.

5 EVALUATION

In this section, we explain the performance characteristics of FHE and demonstrate the overhead that one can expect when using it. Then we demonstrate how the complexity of an application impacts the real-world runtime when using FHE, using the example applications from Section 4 as illustrations.

	Encrypt		Decrypt		Ciphertext size	
	[time]	[op/s]	[time]	[op/s]	[Byte]	[factor]
AES-128	$97ns$	10.8M	$90ns$	10.8M	16	1
RSA-2048	$14\mu s$	71.5k	$0.5ms$	2.0k	256	1.04
FHE - light	$1.7ms$	586	$1.2ms$	827	75k	1.5×10^5
FHE - medium	$1.7ms$	586	$12ms$	87	9.8M	1.2×10^5
FHE - heavy	$66ms$	15	$46ms$	22	9.6M	4.3×10^4

Table 1: Overview of FHE crypto operations compared to symmetric-key based encryption AES-128 and the asymmetric encryption scheme RSA. For FHE, we differentiate between three parameter settings light, medium and heavy. This corresponds to support for 60, 600, or 1800 SIMD-elements per ciphertext, respectively. Using the metric of operations/second (op/s), it is possible to easily relate the performance of FHE to more efficient non malleable encryption schemes. The ciphertext expansion factor describes the ratio between the size of a ciphertext and the maximum information (bits) that can be encrypted within one ciphertext.

5.1 Encryption Overhead

Modern FHE schemes are significantly faster than the initial constructions, allowing real-world use. However, FHE remains expensive even when compared to traditional public-key cryptography. Table 1 gives an overview of the encryption/decryption time and the ciphertext expansion compared to non-FHE public key cryptography (RSA) and symmetric-key cryptography (AES). There is a similar orders-of-magnitude drop in operations/second between FHE and standard asymmetric crypto as there is between asymmetric and symmetric crypto. For FHE, the performance strongly depends on the choice of parameters. For a given security level, larger parameters are required to support more complex computations (i.e., higher multiplicative depth) or to provide more SIMD elements per ciphertext. The *light*, *medium* and *heavy* parameter sets correspond to settings supporting 60, 600 or 1800 SIMD-elements, respectively. While the ciphertext expansion factor is much higher for FHE than it is for standard crypto, it decreases for larger parameters, as ciphertexts can hold more information using the increased number of available SIMD-slots. HElib is used for all FHE evaluations since it is the most appropriate library for the sample applications we presented. The evaluations were performed on a desktop PC with an Intel i7 hexacore CPU and 32 GB of RAM.

5.2 Homomorphic Operations

The fundamental operations in FHE are homomorphic addition and multiplication of encrypted numbers. These operations are significantly slower than the corresponding operations on plaintexts. In Figure 3, we observe several factors that influence the runtime of the operations: The runtime for both operations is significantly larger for parameter settings with many SIMD-slots (heavy) than for ones with few slots (light). In addition, operations for all parameter sets become increasingly expensive as the multiplicative depth

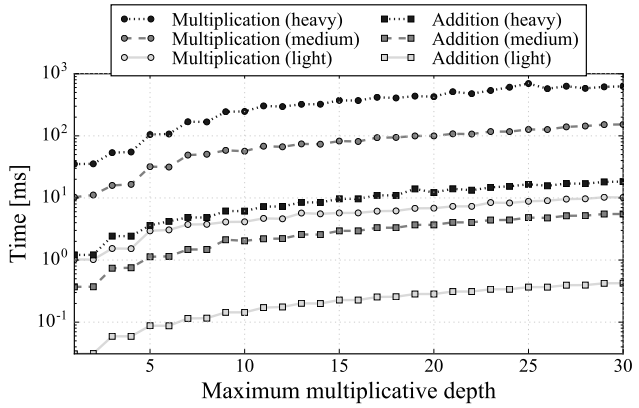


Figure 3: Runtime of the two fundamental homomorphic operations with HElib as the backend crypto engine of Marble, with different parameters. The settings light, medium, and heavy represent the parameter sizes required for SIMD operations with 60, 600 and 1800 elements per ciphertext, respectively. Multiplications are more expensive than additions for any given setting. Also multiplications are slowest when the supported multiplicative depth is highest.

increases. This is because supporting larger multiplicative depth requires larger ciphertexts. Therefore, it is generally advisable to optimize computations to minimize multiplicative depth, even if this results in a slightly higher number of total multiplication operations. Finally, we can see that for each respective parameter setting, multiplications are significantly slower than additions. This is a characteristic of all current FHE schemes, as described in Section 2. Therefore, additions are often considered *free* as their impact on the overall runtime of a computation is negligible.

5.3 Sample Applications

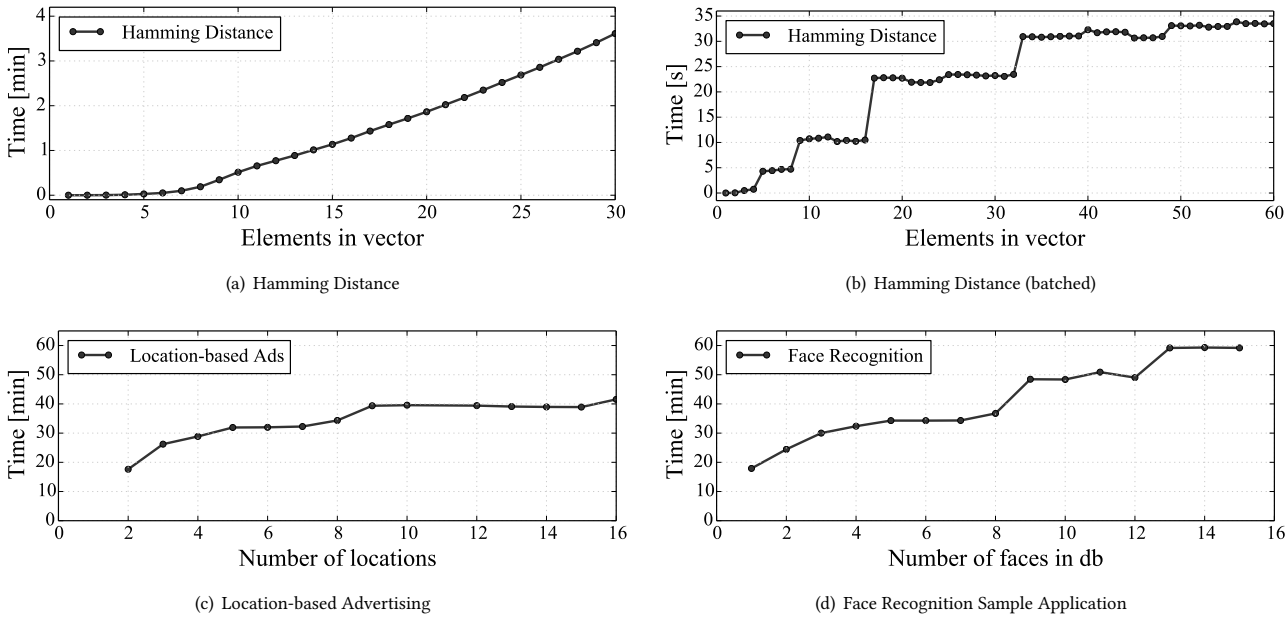
In Section 4, we presented several applications to illustrate how developers can use Marble. Here, we show how the performance of the applications evolves as the applications grow in complexity. All applications use bit-slicing, such that a multiplication between two `M` objects (i.e., the type defined by Marble) corresponds to many homomorphic multiplications that emulate binary gates. All evaluations were carried out with at least 80 bits of security (as reported by HElib's security level estimation function).

Hamming Distance. The first implementation of the hamming distance, `hd_enc` used a vector of values and a for-loop to calculate the differences and the sum. The optimized version, `hd_batched`, instead made use of the SIMD capabilities offered by FHE. With this, the calculation of the differences was reduced to a single operation and the sum was implemented using a SIMD-optimized routine. However, to be able to batch many elements into a single ciphertext the parameter size needs to be increased. In Figure 4(a), we observe the runtime of the non-batched version and in Figure 4(b) the SIMD version, both as a function of the number of elements in the vectors. Even with the larger parameters, the SIMD version significantly outperforms the unoptimized version. This is because without batching, the number of binary arithmetic additions to compute increases linearly with the number of elements, whereas in the batched version the SIMD summation uses a divide-and-conquer approach which increases only logarithmically. We can observe distinct steps in the computation time corresponding to the powers of two. These are caused by the divide-and-conquer approach of the SIMD minimum function. As a result, we can increase the number of elements from, e.g., $n = 17$ to $n = 32$ for virtually no additional cost.

Location-based Advertising. The location-based example includes significantly more complex operations, especially multiplication between M values which is translated to a binary arithmetic circuit with many gates, each corresponding to one homomorphic operation. Figure 4(c) depicts the runtime of the application as a function of the number of places, i.e., the number of elements batched into each ciphertext. Instead of a simple 1-bit comparison, the initial phase now consists of the computation of the squared euclidean distance. With this, we can clearly distinguish the two components of the overall runtime: While the initial phase happens for all locations at the same time, taking advantage of efficient SMID operation, there is still an increase in runtime as the number of locations increases. This is because, as the overall complexity of the computation (especially the multiplicative depth) increases, the parameters have to increase, too. This causes all homomorphic operations to be somewhat slower, as described in Section 5.2. During the second phase, where we compute the minimum index, the number of steps required to compute the final minimum and its index increases logarithmically with the number of locations. However, because the first phase is now significantly more expensive than in the previous example, the logarithmic steps are less distinct in the overall runtime.

Face Recognition. Finally, we discuss the performance of the face recognition application as a function of the number of faces in the database to check against, as depicted in Figure 4(d). The computation is effectively a higher-dimensional ($d = 128$ instead of 2) version of the location-based advertising example. The major difference is that rather than batching each dimension into a separate ciphertext, all dimensions of all inputs are batched into a single ciphertext. With this, we can more efficiently use the available SIMD slots. We note that for $n = 1$, we are performing face-*verification* rather than face-recognition. In face-verification, the task is to check whether or not the input represents a given face. Therefore, for this case we return the computed distance, rather than the minimum index. Again, as in the previous example, the runtime is determined by two main components: increases in parameter size and the (logarithmic) number of steps required to compute the minimum. For $n = 13$ we can observe this clearly: Even though the number of steps required to compute the minimum stays constant, we see an increase in runtime because the number of slots needed exceeds those provided by the previous parameter set, resulting in a larger parameter set and increased runtimes.

While we saw in the batched hamming distance example that increasing the number of elements up to the next power of two was virtually free, this is only true for this example as long as the increase in elements does not cause a switch to a significantly larger parameter set.



(a) Hamming Distance

(b) Hamming Distance (batched)

(c) Location-based Advertising

(d) Face Recognition Sample Application

Figure 4: Runtime of the sample applications as a function of the input size. The direct Hamming Distance implementation takes 30s for 10 elements. In the same time, the batched version can process a vector with 32 elements. This is because the batched version only requires logarithmic depth to add all values. For the location-based advertising example which contains many more homomorphic operations, the added runtime due to increased parameters is more noticeable and the logarithmic contribution of the minimum function is less prominent. The same is true for the face recognition example, e.g., the bump at $n = 13$ is caused by a change in the parameter set required to accommodate the increased number of slots needed.

6 IMPLEMENTATION

Marble is written in modern C++, making use of features introduced in C++11. The Marble library currently includes roughly 2.3k LOC. This section outlines the different parts of the library, how they interact conceptually, and how they are implemented.

6.1 Interacting with user code

The public interface of Marble is defined by a single class `M` and a few non-member utility functions (primarily `analyze` and `evaluate`). Developers can use `M` like a built-in numeric type (e.g., `int`), allowing code that looks very similar to plaintext applications, as demonstrated in Section 4. Should a part of the code that contains Marble objects be executed directly, rather than via `analyze` or `evaluate` then Marble simply behaves like a plaintext type. Developers specify an entrypoint for their FHE application by passing a function returning `void` to `analyze` or `evaluate`. With this, developers are not limited to the default C++ entry point (`main`), which makes it easier to evaluate multiple FHE applications in a single project. The formal parameter type for both functions is `std::function<void()>`, i.e., a function that takes no arguments and does not return anything. In order to nevertheless use functions that have arguments, developers can use `std::bind()` to bind the arguments to the function (essentially partial application), as demonstrated in the examples in Section 4.

Supporting a clean public interface requires some behind-the-scenes accounting to allow developers to write their code once and have Marble either remain passive, analyze the computation, or evaluate it using FHE. This is achieved via a set of static members in the class `M` that track whether or not the library is supposed to be active and if yes, which mode to operate in. Similarly, the results of the analysis are stored in static class members in order to enable users to create, destroy, copy and pass individual `M` objects without consideration for the underlying analysis.

6.2 Analysis and Evaluation

When called via `analyze`, Marble performs a dry-run simulation of the computation without any FHE. It uses this to determine the type and size of (encrypted) inputs, the operators used during the computation and the maximum multiplicative depth. From this, it generates and prints a report about the complexity of the application. When the system is called via `evaluate`, the analysis data is used to determine the choice of FHE library, encoding and parameters to use. When comparison operations are used (i.e., when bit-slicing is necessary), Marble prefers HElib because of its built-in support for efficient binary arithmetic circuits [12]. Marble uses a custom fork of HElib with some extensions to efficiently apply these circuits to `fold` style operations. In a similar vein, SEAL is preferred when the inputs are fractional numbers since SEAL provides a built-in fractional encoder [9].

The analysis works internally by calling the user-supplied function but the operators in the `M` class perform no homomorphic operations and instead only update the complexity measures.

Since the system can only decide which library or encoding will be used after the analysis, the analysis has to account for all possibilities. E.g., we calculate the complexity both with and without bitslicing. When the analysis encounters an operation that is only possible with a specific encoding, it sets the corresponding (static) flag. Finally, the system uses the calculated complexity and the flags to choose an appropriate library, encoding, and set of parameters for evaluation. The parameters are chosen by first searching for the best match in a list of known "good" parameter sets for each library. Should none be found, then we can fall back to HElib's or SEAL's parameter-search functions. However, if none of the default parameter sets is applicable this already indicates that the application is potentially too complex.

Marble has two ways of communicating the limitations of FHE to developers. First, there are compile-time effects that represent the semantic limitations of FHE, e.g., the inability to branch execution based on encrypted values is enforced by the fact that no valid conversion from `M` types to `boolean` exists. The second way of providing feedback happens at run-time: During analysis, Marble warns if the size of the inputs, or the multiplicative depth of the computation are high enough to make an evaluation infeasible.

6.3 FHE Back-end

Marble is designed to support HElib and SEAL as the back-end FHE implementations, but could easily be extended to support other libraries. We discussed our reasoning for choosing these two libraries in Section 3, but to summarize they currently represent the most mature implementations of the two most established FHE schemes. Both libraries offer very different interfaces for the same operations, therefore we introduce a layer of abstraction via wrapper classes to arrive at a common abstraction for encrypted numbers. Objects of type `M` are associated during evaluation with the underlying FHE objects via a `std::unique_ptr` to a wrapper class. These wrappers also provide associated utility functions to handle set-up and instantiation, as both libraries create large data-structures during set-up which are then used throughout the computation. Marble already supports a large range of operations using HElib, as demonstrated in Section 4. The SEAL-based implementation, meanwhile, is under construction.

7 RELATED WORK

The following section introduces the most prominent alternatives to FHE and their advantages and disadvantages. In this section, we put our framework into the context of existing and proposed FHE tools. Additionally, we discuss prominent alternatives to FHE and their advantages and disadvantages.

FHE Tools

HEide [17] is a python-based graphical IDE for HElib that aims to make working with the library easier. However, it supports only a fraction of the library's features and does not offer a higher-level abstraction (i.e., users must implement circuits). Therefore, the performance of an application implemented this way is not indicative of how well this application could be implemented using FHE.

Armadillo [5], on the other hand, is closer to our work. It presents a compiler for a subset of C++ programs, converting them to FHE circuits. The system supports only bit-wise encryption, though admittedly this is also by far the more complex case to handle for a framework. We nevertheless consider it valuable to have a tool that supports both bit-wise and non-bitwise FHE, to allow developers to quickly switch between the two settings. This helps developers to decide whether they need the computationally expensive bit-wise encryption and the associated expressiveness, e.g., comparison operations. In addition to this, Armadillo is limited to integer variables, which unnecessarily restricts what applications can be realized.

In Armadillo, the C++ program is first translated into a circuit using trivial arithmetic circuits (rather than ones optimized for FHE, as in our case). In a second step, these circuits are then optimized using an optimization tool borrowed from the hardware domain. Circuit optimizations for physical implementations do not necessarily carry over to the domain of FHE. Hence, they resort to using a simple heuristic for circuit optimization. Nevertheless, this approach seems promising, especially assuming that general circuit-optimization techniques from MPC might prove to be a better match for FHE than hardware circuit optimization tools.

Alternatives to Bit-Wise Encryption

In this paper, we used bit-wise encryption when we considered cases where we needed non-polynomial functions (e.g., comparison). However, it is also possible to instead approximate these functions as polynomials. Çetin et al. present a variety of approximations of inversion, comparison and other useful operators in [8].

Hand-written FHE Applications

Orthogonally to our work, there are several systems employing FHE that were hand-crafted by experts. These employ a variety of advanced techniques that can be used to further enhance the performance of an FHE-based application. However, they require to be adapted specifically to each target application. Moreover, they exhibit a high level of insight into the underlying FHE schemes and are therefore highly non-trivial for non-experts to implement.

Some systems gain a performance benefit from using lower-level features of the underlying cryptosystem: For example, some problems map well to more complex plaintext spaces (e.g., $\mathbb{F}_{p^k}$), as demonstrated in the homomorphic implementations of the AES encryption circuit by Smart et al. [38]. Working directly with the hypercube view of SIMD slots and choosing input sizes to match with the various native dimensions can also help to reduce the number of homomorphic multiplications needed, as shown by the FHE-based implementation of logistic regression by Crawford et al. [12].

Alternatively, some works employ comparatively simple FHE techniques but apply complex transformations on the problem itself to make it more suitable for the FHE evaluation: Grapel et al., for example, show how to perform machine learning on encrypted data using FHE. They achieve this by constructing special polynomial approximations of well-known machine learning algorithms that are more suitable for FHE applications [22]. Wang et al. also present a system to privately outsource feature-extraction to the cloud using

FHE by redesigning the Histogram-of-Gradients calculation to be more suitable for FHE [40].

Combining FHE with other privacy preserving techniques, such as interactive protocols, can lead to interesting hybrid solutions: Çetin et al. present a system that can evaluate queries on encrypted genomic data that combines techniques from FHE with private set interaction protocols [7]. Boneh et al. construct a multi-party protocol for private information retrieval on databases and show that a version using FHE can outperform the one using partially homomorphic encryption [2].

Secure Multi-Party Computation

FHE is similar to secure Multi-Party Computation (MPC), which also allows computing on encrypted data. Historically, MPC is seen as an n-party problem where the parties want to compute a shared result while keeping their inputs secret. However, MPC can also reveal the result to only a subset of the parties, similar to the setting considered in FHE. The main difference between the two approaches is that MPC generally has low computation costs but high communication costs, usually requiring a large amount of interactions while FHE has high computation costs but low communication costs and computations can be performed offline. This means FHE aligns well with the current trend of outsourcing storage and computation to the cloud, especially in mobile and web-based applications. MPC is, however, gaining more traction within the decentralized storage community, to provide added value on data of multiple users, without disclosing any of the inputs [42, 43].

In contrast to FHE, where most scenarios consider a client and a semi-honest server, MPC solutions make use of a wide variety of trust models. This includes trust models where one assumes that two or more servers do not collude with each other, but does not trust the individual servers. In such models, high performance can be achieved [28].

Since MPC is a mature research field, a variety of systems exist that address a large array of challenges, such as private information retrieval or outsourcing computations. For example, privacy-preserving face-recognition using MPC was proposed as early as 2006 [1]. Because of the significantly lower computation requirements, MPC systems can even be used on mobile phones in many scenarios [13, 27, 29]. The main challenge for FHE-based implementations is therefore to reduce the client-side computation to a point where the offline-computation abilities of the server and the reduction in interactivity outweigh the increased client-side computation. It is even possible to see FHE and MPC as two extremes on a continuum, moving from total interactivity to total offline computation, with the optimal balance depending on the application [10].

Since MPC computations are usually defined via circuits, there is the potential of applying optimization techniques from MPC to FHE circuits, This is further reinforced by the fact that in MPC, additions (XOR) are also *free* when compared to multiplications (AND) [26]. This means that using circuit optimization techniques developed for MPC could be a more promising avenue than trying to adapt circuit synthesis algorithms from the hardware domain to the FHE setting.

Partially Homomorphic Encryption

While partially homomorphic encryption (PHE) schemes do not offer the same expressiveness as FHE, systems combining partially homomorphic encryption schemes with other encryption schemes, such as order-preserving encryption, can also achieve well defined computations in certain applications, such as computing aggregates as one of the most common database queries. PHE schemes are generally faster and more practical to use, as several encrypted database systems [31, 36, 37, 39] have shown. The most popular additively homomorphic encryption scheme is the Paillier cryptosystem [30].

The Paillier cryptosystem explicitly improves the ciphertext expansion problem of the previous schemes, such as Goldwasser-Micali's scheme [21] which was among the first additive homomorphic encryption schemes achieving semantic security. With 80 bit security, Paillier encryption takes an input of up to 1024 bit and generates a 2048 bit ciphertext [37]. Known multiplicatively homomorphic encryption schemes are unpadded RSA [33] and ElGamal [14]. The elliptic curve version of ELGamal is also explored in encrypted databases [36, 37] as an alternative to Paillier, with more efficient encryption and decryption operations, due to operating over smaller key sizes compared to the Paillier cryptosystem. Somewhat homomorphic encryption schemes typically allow an unlimited number of additions and few multiplications, such as the BGN scheme [3] or constructions that transform a linearly-homomorphic encryption into a scheme capable of evaluating degree-2 computations [6]. However, some of these schemes [6] violated the compactness requirement, such that the ciphertext size grows to hold state about the operations to be completed after decryption.

8 CONCLUSION

The field of Fully Homomorphic Encryption (FHE) has made significant advancements in the recent years. However, the entrance barrier to explore the potential of FHE-based applications is still too high and this deters curios researchers and developers. With Marble we make an attempt to alleviate this by providing a more familiar development environment and abstracting away the complexity of setting up the correct parameters of the underlying FHE scheme. Marble supports researchers and developers without deep understanding of the cryptography behind FHE to rapidly develop an FHE-based prototype of their algorithms, assess its feasibility, and gain an overview of the overall performance of their FHE-based application. Marble leverages two of the most promising FHE libraries in the backend, and has the possibility to employ more advanced and efficient ones as they are developed by the community of cryptographers. We make Marble available as a convenient ready-to-use docker image and hope to facilitate easier access to exploring the potential of FHE-based applications.

REFERENCES

[1] Shai Avidan and Moshe Butman. 2006. Blind Vision. In *Computer Vision – ECCV 2006 (Lecture Notes in Computer Science)*, Aleš Leonardis, Horst Bischof, and Axel Pinz (Eds.). Springer, Berlin, Heidelberg, 1–13. https://doi.org/10.1007/11744078_1

[2] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J Wu. 2013. Private Database Queries Using Somewhat Homomorphic Encryption. In *Applied Cryptography and Network Security (Lecture Notes in Computer Science)*,

Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer, Berlin, Heidelberg, 102–118. https://doi.org/10.1007/978-3-642-38980-1_7

[3] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. 2005. Evaluating 2-DNF Formulas on Ciphertexts. In *Theory of Cryptography*. Springer, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/978-3-540-30576-7_18

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (July 2014), 13:1–13:36. https://doi.org/10.1145/2633600

[5] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing (SCC '15)*. ACM, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[6] Dario Catalano and Dario Fiore. 2015. Using Linearly-Homomorphic Encryption to Evaluate Degree-2 Functions on Encrypted Data. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1518–1529. https://doi.org/10.1145/2810103.2813624

[7] Gizem S Çetin, Hao Chen, Kim Laine, Kristin Lauter, Peter Rindal, and Yuhou Xia. 2017. Private queries on encrypted genomic data. *BMC medical genomics* 10, Suppl 2 (July 2017), 45. https://doi.org/10.1186/s12920-017-0276-z

[8] Gizem S Cetin, Yarkın Doröz, Berk Sunar, and William J Martin. 2015. Arithmetic using word-wise homomorphic encryption. (2015). https://eprint.iacr.org/2015/1195.pdf

[9] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple Encrypted Arithmetic Library - SEAL v2.1. In *Financial Cryptography and Data Security*. Springer International Publishing, 3–18. https://doi.org/10.1007/978-3-319-70278-0_1

[10] Ashish Choudhury, Jake Loftus, Emmanuela Orsini, Arpita Patra, and Nigel P Smart. 2013. Between a Rock and a Hard Place: Interpolating between MPC and FHE. In *Advances in Cryptology - ASIACRYPT 2013 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 221–240. https://doi.org/10.1007/978-3-642-42045-0_12

[11] Ana Costache and Nigel P Smart. 2016. Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?. In *Topics in Cryptology - CT-RSA 2016 (Lecture Notes in Computer Science)*. Springer, Cham, 325–340. https://doi.org/10.1007/978-3-319-29485-8_19

[12] Jack L H Crawford, Craig Gentry, Shai Halevi, Danil Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. (2018). https://eprint.iacr.org/2018/202.pdf

[13] Emiliano De Cristofaro, Sky Faber, Paolo Gasti, and Gene Tsudik. 2012. Genodroid: Are Privacy-preserving Genomic Tests Ready for Prime Time?. In *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society (WPES '12)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2381966.2381980

[14] T Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory / Professional Technical Group on Information Theory* 31, 4 (July 1985), 469–472. https://doi.org/10.1109/TIT.1985.1057074

[15] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. 2009. Privacy-Preserving Face Recognition. In *Privacy Enhancing Technologies (Lecture Notes in Computer Science)*, Ian Goldberg and Mikhail J Atallah (Eds.). Springer, Berlin, Heidelberg, 235–253. https://doi.org/10.1007/978-3-642-03168-7_14

[16] Gizem S Çetin, Wei Dai, Yarkın Doröz, and Berk Sunar. 2015. Homomorphic Autocomplete. Cryptology ePrint Archive, Report 2015/1194. http://eprint.iacr.org/2015/1194

[17] Grant Fame. 2015. *HEIDE: An IDE for the Homomorphic Encryption Library HElib*. Master's thesis. California Polytechnic State University, San Luis Obispo. http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=2523&context=theses

[18] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802

[19] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph.D. Dissertation. Stanford University. https://crypto.stanford.edu/craig/

[20] Craig Gentry. 2010. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* 53, 3 (March 2010), 97–105. https://doi.org/10.1145/1666420.1666444

[21] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC '82)*. ACM, New York, NY, USA, 365–377. https://doi.org/10.1145/800070.802212

[22] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *Information Security and Cryptology – ICISC 2012*. Springer Berlin Heidelberg, 1–21. https://doi.org/10.1007/978-3-642-37682-5_1

[23] Shai Halevi and Victor Shoup. 2013. *Design and implementation of a homomorphic-encryption library*. https://github.com/shaih/HElib/blob/master/doc/designDocument/he-library.pdf

[24] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014 (Lecture Notes in Computer Science)*, Juan A Garay and Rosario Gennaro (Eds.). Springer, Berlin, Heidelberg, 554–571. https://doi.org/10.1007/978-3-662-44371-2_31

[25] Shai Halevi and Victor Shoup. 2015. Bootstrapping for HElib. In *Advances in Cryptology – EUROCRYPT 2015 (Lecture Notes in Computer Science)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Berlin, Heidelberg, 641–670. https://doi.org/10.1007/978-3-662-46800-5_25

[26] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, Berlin, Heidelberg, 486–498. https://doi.org/10.1007/978-3-540-70583-3_40

[27] Payman Mohassel, Ostap Orobets, and Ben Riva. 2016. Efficient Server-Aided 2PC for Mobile Phones. *Proceedings on Privacy Enhancing Technologies* 2016, 2 (Jan. 2016), 378. https://doi.org/10.1515/popets-2016-0006

[28] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 19–38. https://doi.org/10.1109/SP.2017.12

[29] Benjamin Mood, Lara Letaw, and Kevin Butler. 2012. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 254–268. https://doi.org/10.1007/978-3-642-32946-3_19

[30] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99 (EUROCRYPT)*. Springer, Berlin, Heidelberg, Prague, Czech Republic, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[31] Raluca Ada Popa, Catherine M S Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 85–100. https://doi.org/10.1145/2043556.2043566

[32] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180. https://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf

[33] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342

[34] Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2009. Efficient Privacy-Preserving Face Recognition. In *Information, Security and Cryptology – ICISC 2009 (Lecture Notes in Computer Science)*, Donghoon Lee and Seokhie Hong (Eds.). Springer, Berlin, Heidelberg, 229–244. https://doi.org/10.1007/978-3-642-14423-3_16

[35] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 815–823. https://doi.org/10.1109/CVPR.2015.7298682

[36] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. 2017. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. ACM, New York, NY, USA, 29:1–29:14. https://doi.org/10.1145/3131672.3131697

[37] Hossein Shafagh, Anwar Hithnawi, Andreas Droescher, Simon Duquennoy, and Wen Hu. 2015. Talos: Encrypted Query Processing for the Internet of Things. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 197–210. https://doi.org/10.1145/2809695.2809723

[38] Nigel P Smart and Fréderik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography. An International Journal* 71, 1 (April 2014), 57–81. https://doi.org/10.1007/s10623-012-9720-4

[39] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases*, Vol. 6. VLDB Endowment, 289–300. https://doi.org/10.14778/2535573.2488336

[40] Qian Wang, Jingjun Wang, Shengshan Hu, Qin Zou, and Kui Ren. 2016. SecHOG: Privacy-Preserving Outsourcing Computation of Histogram of Oriented Gradients in the Cloud. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/2897845.2897861

[41] Can Xiang, Chunming Tang, Yunlu Cai, and Qiuxia Xu. 2016. Privacy-preserving face recognition with outsourced computation. *Soft Computing* 20, 9 (Sept. 2016), 3735–3744. https://doi.org/10.1007/s00500-015-1759-5

[42] G Zyskind, O Nathan, and A Pentland. 2015. Decentralizing Privacy: Using Blockchain to Protect Personal Data. In *2015 IEEE Security and Privacy Workshops*. 180–184. https://doi.org/10.1109/SPW.2015.27

[43] Guy Zyskind, Oz Nathan, and Alex Pentland. 2015. Enigma: Decentralized Computation Platform with Guaranteed Privacy. arXiv (whitepaper) http://www.enigma.co/enigma_full.pdf.