Techniques, Software, and Applications for Packed Partially Homomorphic Encryption

Mikhail Volkhov*

The University of Edinburgh, UK
mikhail.volkhov@ed.ac.uk

Introduction

Secure Machine Learning analyses different ML solutions trying to answer the question of how to make these algorithms conform nowadays security and privacy requirements. Most of the solutions fall into the category of either secure training, or secure inference (classification or regression). The active research on fully homomorphic encryption (FHE) that started right after Gentry's [Gen09] proposal together with the latest improvements in protocol design produced a lot of different secure computation models that allow to easily adapt existing ML methods, like classifiers, to meet the common security goals. One of the popular targets to apply these techniques to are Convolutional Neural Network classifiers, and recently it was shown that it is possible and practical to execute CNN linear layers using modern FHEs [JVC18]. On the other hand, a number of different techniques that make use of simpler classifiers, like SVM, can achieve comparable classification accuracy, using much simpler cryptographic primitives [MRSV19]. Different combinations of multiparty protocols, homomorphic encryption, and classifiers present many open problems related to the performance, accuracy, and security. Verification is another concern — many solutions are implemented only as prototypes and do not take the systematic verified approach that is possible to introduce with software verification techniques.

The Research Question

The research direction we take is investigating the combination of the modern FHE-inspired techniques that allow to perform complex tasks homomorphically (e.g. linear algebra operations — matrix multiplications or convolutions) with simpler, more robust and well-studied partially homomorphic schemes (PHEs). Our main target was to take the advantage of both high security guarantees we get with F* language and, on the other hand, performance and functional guarantees of the newer solutions. The problem of vectorizing (or packing) PHE encryption was not studied in detail before, and all the related applications are using standard naive methods to achieve much simpler parallelism functionality. Different protocols can take the advantage of these parallelization techniques, decreasing computation and communication complexity.

The Contribution

Our findings show that it is possible to achieve plaintext packing functionality for the partially homomorphic encryption scheme called DGK [DGK08, DGK09], and we present two packing modes to do that. These packing modes support parallel addition and multiplication, but we were not able to discover a way to perform plaintext space permutations — the operation that

* This work was done as part of MPRI Master's thesis during the internship in PROSECCO, INRIA Paris, under the supervision of /Karthikeyan Bhargavan and Prasad Naldurg; August 2019.

is required for homomorphic matrix operations. Nevertheless, the packed operations are enough to parallelize a different set of simpler algorithms that form the basis for SVM and Naive Bayes classifiers. Namely, we improve integer comparison and argmax protocols. We also introduce a new protocol called LogArgmax, that takes advantage of batched comparisons in order to significantly decrease number of communication rounds.

We implement three encryption schemes — Paillier, Goldwasser-Micali and DGK encryption, in F* language, as part of the verified cryptographic library called HACL*. We provide specification-level lemmas about functions, and implementation we have is fully verified and memory safe with respect to the specification. We implement packing methods and protocols, as well as benchmarking tools in Haskell, using bindings to C code generated from F*.

Arguments Supporting Validity of the Contribution

Our practical results show that the batched variants of the protocols indeed give better performance results than their non-batched variants, especially in high-latency environments. We also show that the LogArgmax algorithm is more effective than its linear counterpart. Finally, we provide a microbenchmarks of the implemented cryptographic primitives that show that they are fast enough to be used in most of the applications (although optimizing the primitives was not a primary objective of this work).

Summary and Future Work

We have shown that it is possible to use PHEs in a packed way somewhat similar to FHEs, and that it has positive practical implications. There exist several immediate steps that can be taken to improve the protocols further, including even more effective use of batched comparison in LogArgmax algorithm. The F* implementation can provide more functionality, including key generation, which is currently done outside of F*. Moreover, the performance of PHEs depend on the underlying long arithmetic library, which can be improved both in terms of performance and verification (which is a further work direction).

As we do not provide a proof of permutation impossibility in PHEs, but rather merely show that our packing modes do not allow it, the general possibility question is still open. If the answer to it is positive, it will be possible to build an abstract linear algebra implementation and compare its performance on the real CNN evaluation with the existing FHE-based results.

1 Background

This section gives a short overview of the problem studied and of the concepts we are using to formulate the main research question.

1.1 Homomorphic Encryption

Homomorphic encryption (HE) schemes are encryption schemes that allow operations on plaintexts such that the encryption function is homomorphic with respect to these operations. This means that functions composed of arithmetic operations can be executed by the third party on the encrypted values, without learning any information about plaintexts inside. The most common operations are modular addition and multiplication. Older HE schemes such as Paillier [Pai99], Damgård-Jurik [DJ01], DGK [DGK08, DGK09], GM [GM84] (also called QR for "quadratic residues"), and others are using well-understood algebraic mechanics, and well-studied assumptions such as finite field DLPs, quadratic residuosity, and RSA-related factoring problems. Usually, in these schemes only one truly homomorphic operation is available — addition or multiplication. These schemes are called partially homomorphic (PHEs). Schemes that carry a plaintext message in the exponent (all mentioned above except for GM) naturally represent homomorphic addition as multiplication of ciphertexts (by summing the exponents), but some of them also allow to perform multiplication by a scalar plaintext value v by raising the ciphertext to the power v. Partially homomorphic schemes whose main operation is additive are called additively homomorphic (AHEs), and they will be the main focus in this work.

Since Gentry's discovery [Gen09] a number of fully homomorphic encryption schemes (FHEs) appeared, allowing to perform arbitrary number of both additions and multiplications. At first, this came at a cost of worse performance, but more recent schemes are getting closer to the practical application requirements. Some FHEs have extra properties that can be used in specific application areas. For example, FHEs that are based on the BGV [Bra12, BGV14] or FV scheme [FV12], use polynomial plaintext space $\mathbb{Z}_p[X]/\Phi_m(X)$. Because of this, and with the help of several additional algebraic methods, they can encrypt vectors of $\mathbb{Z}_p$ elements, so that both homomorphic addition and multiplication of these vectors is component-wise, or, in other words, SIMD (single instruction multiple data). Even more, these schemes (and others, for instance [SV14, SV10]) can perform rotations, element swaps, and permutations homomorphically. In fact, a possibility to do rotations is enough to obtain the rest of the operations, as is shown in the work by Gentry et al. [GHS12]. FHEs are now thoroughly researched and improved with respect to the security and performance balance. For more details on homomorphic encryption schemes in general see [AAUC18, ALW16].

From the implementation perspective, there exist a number of libraries that provide different modifications of modern FHEs. HELib [HS] provides the implementation of BGV scheme, including the linear algebra code making use of the SIMD operations, rotations and permutations mentioned [hal14]. Another popular library is Microsoft SEAL [SEA19] which implements two FHEs — FV and HEAAN [CHK+18] (which is another popular FHE). PHE schemes are much less covered — Paillier (and its generalisation DJ), and GM have many open-source implementations, but none of them seem to be properly reviewed or audited and thus serve as a reference implementation.

1.2 Machine Learning Applications

A particularly interesting and popular application of homomorphic schemes is secure evaluation of machine learning algorithms. The concrete algorithms are numerous — naive Bayes

classifiers, tree classifiers [BPTG15], SVMs [MRSV19, RWT+18], linear and logistic regressions [MZ17, KSW+18], neural networks [MZ17, RWT+18, RRK18, WGC18, JVC18, GBDL+16, LJLA17, CBL+18] — including convolutional neural networks (CNNs), and more. Approaches taken are various, but usually they make use of multiparty protocols (some with HE-based setup) [RWT+18, MZ17, RRK18, WGC18, MRSV19, LJLA17], homomorphic encryption [JVC18, BPTG15, GBDL+16, CBL+18, KSW+18], or both combined. Most of the HE-inspired solutions are using FHEs, based on the implementations covered previously. The work by Bost et al. [BPTG15] uses PHE as a basis for their two party computation (2PC) algorithms. We will now describe several of the mentioned solutions in more detail.

The most common pattern of HE-based secure ML solutions is the following one: one party holds the secret key and performs most of the HE computations, while the other one may or may not engage in supporting 2PC computation related to the main homomorphic computation. We can imagine client owning the secret key, and sending its encrypted input to the server, where it is transformed homomorphically into the encrypted output and then sent back to the client, where the answer is then decrypted and observed. It was shown in Gazelle [JVC18] that it is possible to evaluate a plaintext CNN model (more precisely, its linear layers) homomorphically in this client-server scenario using only three types of operations: SIMD addition, SIMD multiplication by a plaintext scalar, and slot permutation. Linear layers are matrix convolutions, and matrix-matrix or matrix-vector multiplications, where matrices and convolution kernels are known to the server. Evaluation of non-linear functions (corresponding to non-linear CNN layers), such as ReLU, sigmoid, or max pooling functions, can be done by approximating them with polynomials and evaluating them with FHE or some 2PC such as Yao's Garbled circuits [Yao86]. Gazelle takes the second path, while an example of the first approach is CryptoNets [GBDL+16, CBL+18]. Non-linear function evaluation is somewhat separate research area and is not discussed further in this document.

The ability to evaluate linear algebra operations homomorphically gives a lot of flexibility in terms of transferring existing ML solutions to the secure evaluation realm, but this is not the only approach taken. Alternatively, one can use simpler ML algorithms that are easier to adapt to HE evaluation requirements. For instance, EPIC [MRSV19] uses anonymous and publicly available feature extraction NN to train a linear SVM classifier that, combined with the feature extraction, achieves better accuracy than other solutions, like Gazelle and MiniONN [LJLA17]. SVM classification algorithm is much simpler to evaluate: $\mathsf{result}(y) = \arg\max_i(x_i \cdot y + b_i)$, where x_i are SVM support vectors, b_i is a model intercept (both are secrets that belong to the server-stored model), and y is client's input. Because of this, EPIC also achieves significantly better performance than aforementioned works.

While EPIC uses a modification of SPDZ 2PC [DPSZ12] by Damgård et al. (its variations are very common in 2PC solutions) to perform a dot product and argmax computation, the work by Bost et al. [BPTG15] uses a mix of 2PC and PHE to achieve the same goal. It combines Paillier, GM, and DGK computations with different 2PC techniques (e.g. additive blinding and server-aided plaintext manipulation) and comparison protocols to build the two-party argmax algorithm and several classifiers — private hyperplane decision, naive Bayes, and decision tree. The first two classifiers are built on argmax and dot product evaluation.

1.3 Software Verification with F*

F* is a high-level ML-like programming language aimed at program verification and capable of expressing a vast variety of correctness properties [fst]. The language's type system is quite rich, including support for polymorphism, dependent types, refinement types, and a weakest

precondition calculus. The verification is done with the help of SMT solver. After F* program is verified, it can be compiled to C (using tool called kremlin), OCaml, F# and WebAssembly.

HACL* [ZBPB17] is a portable cryptograhic library written in F* and then compiled to C that implements many modern cryptographic primitives. It implements a minimalistic NaCl cryptographic API and can be a drop-in replacement for any other cryptographic library based on the same API, such as LibSodium or TweetNaCl. The performance of HACL* primitives is same as their implementation in pure C and even better in some cases, but still noticeably slower than manually optimised assembly implementations. Given the overall practical trade-off between performance and security guarantees, in this work we will be building on our own implementation of encryption schemes as a part of HACL*.

The two main goals for verification approach are functional correctness and memory safety. Implementations of cryptograhic primitives in F* are usually divided into two parts — specification, written in high-level F* (using high-level primitives, like lists and embedded bignum integers), and implementation, written in Low* subset of F* language, that operates on more primitive concepts like machine integers and buffers. Specifications provide functional correctness guarantee, meaning that the functionality implemented can be proven to do what is theoretically suggested. A simple example of a functional correctness proof for the encryption system is a lemma stating that decryption is the inverse of encryption. Functional correctness also covers the relation between the Low* implementation and specification — precisely, we provide a proof that implementation functions work in the same way as their specification-level counterparts. Memory safety property guarantees that the Low* code does not violate any memory-related assumptions — that it does not read or write to the memory at invalid locations, that we never use freed memory, work with non-allocated memory, etc. This is done by carefully expressing generalised memory models that correspond to the memory models of the languages we compile F* to. HACL* library design also provides a way to write code that is resistant to side-channel attacks. One way is forcing constant-time coding paradigm — restricting the set of operations that can be perform on secret values, therefore including operations that affect observable events, such as branching, power consumption, cache hits and misses, etc.

Formalizing some specification parts in F* can potentially reach the limits of being practical due to the limitations of the prover itself. For instance, F* for now does not have any tools to reason about arbitrary precision real numbers or probability distributions, which means that very few lemmas about such specifications can be proven. Most of the modern FHE schemes are using these concepts in one or another form. For instance, HEAAN [CKKS17, CHK+18] encryption — a promising modern FHE scheme, implemented in SEAL, and partially implemented in HELib — uses both extensively, as well as BGV and FV (which only need probability distributions). These schemes' specifications are also much heavier in terms of complexity of the concepts used, and proving properties on the specification level about them seems to be highly impractical for now, which leaves us with just specification-implementation correctness part of functional correctness proofs. On the other hand, PHE schemes are not only much easier to implement and reason about, but they also lack any verified, or in other way trusted, and fast implementations, and thus they seem to suit well as a target for being implemented from scratch as part of HACL*.

1.4 The Research Question

Given the relatively high complexity of modern FHE schemes, compared to the concrete applications that are not extremely demanding in terms of functionality, we investigate the potential combination of PHEs with the packing functionality of the modern FHEs. We would like to know if it is possible to provide SIMD homomorphic operations required to perform linear algebra at a comparably high functional level as it is done in Gazelle. In other words, the main research direc-

tion of the document is PHE plaintext packing methods, their implementation and application to the secure ML algorithms mentioned.

2 Building Blocks

We shall be using several different encryption schemes, which we briefly cover in this section, and also a number of protocols that we further optimise and use to build our final application-level protocol.

2.1 Additive Partially Homomorphic Encryption Schemes

Our primary goal is to investigate the capabilities of simple PHEs. We use terms AHE and PHE interchangeably since the work does not cover any additive FHEs or multiplicative PHEs.

The Goldwasser Micali (GM) cryptosystem [GM84] is a simple additively homomorphic encryption scheme encrypting bit values. It is based on the quadratic residuosity problem and uses big like composite number as a ciphertext modulus. Paillier cryptosystem [Pai99] is also using RSA-like modulus n and is based on the (higher-level) composite residuosity problem. Its plaintext space is $\mathbb{Z}_n$, and ciphertext space is $\mathbb{Z}_{n^2}$. Both cryptosystems are additively homomorphic and support multiplication by a scalar. The detailed description of Paillier and GM cryptosystems is given in Appendix A.

The particular choice of PHEs is motivated by several factors. First of all, some are them are optimised to work with the particular protocols we will be using — DGK is designed for comparing unencrypted integers, while Paillier fits better for encrypted integers comparison because of the big plaintext size. The second thing is that DGK is the scheme we shall be using for the plaintext packing because of its specific properties, and no other schemes seem to be as suitable. Also, arguably, the schemes we have chosen are among the most used and popular in the secure ML area, which justifies their choice for the verified implementation aspect.

DGK AHE. The particular cryptosystem we would like to analyse in more detail is the one by Damgård, Geisler, and Krøigård [DGK08, DGK09], abbreviated to DGK. A main distinction of DGK compared to Paillier is using a subgroup of $\mathbb{Z}_n^*$ as a plaintext space which shrinks the ciphertext space from $\mathbb{Z}_{n^2}$ to $\mathbb{Z}_n$. As in Paillier, n is a composite RSA-like number. DGK has the same homomorphic properties as Paillier: ciphertext multiplication is a homomorphic addition, and ciphertext exponentiation is a multiplication by scalar. It is also similar to the Groth's cryptosystem [Gro05] — the form of encryption function is the same and decryption is is similar to the one of DGK.

DGK setup is the following. We first pick three primes u, v_p, v_q, and two other random primes r_p and r_q to create RSA modulus $n = (uv_pr_p + 1)(uv_qr_q + 1)$. Primes v_p and v_q are each about $t = 160$ bits and we denote $v = v_pv_q$. We also choose two random elements $g, h \in \mathbb{Z}_n^*$ such that the multiplicative order of h is v both modulo p and q, and multiplicative order of g is uv. The secret key is now (r_p, r_q, v_p, v_q) while the public key is (n, g, h, u). Unlike in Groth's cryptosystem (and several other residuosity-based encryption schemes), the plaintext subgroup size u is public. As we will see later, this property is crucial for our application. The encryption of message $m \in \mathbb{Z}_u$ with random integer r of bit length $2t$ is given by

$$E(m, r) = g^m h^r \mod n$$

Since $E(m, r)^v = (g^v)^m$ (as h cancels out), and the order of g^v is u, we now can solve the discrete log base g^v to obtain m back. To speed the decryption up, we would like u to have many

small factors, so that we can enable Pohlig-Hellman [PH78] style DLP computation. If the order of g is $N = q_1^{e_1} \dots q_t^{e_t}$, and we can solve DLP for some base $\hat{g}$ with order q^e in time $\mathcal{O}(S_{q^e})$, then we can solve DLP for g in $\mathcal{O}(\sum S_{q_i^{e_i}} + \log N)$ steps. A similar reduction for a single q^e order from the naive $\mathcal{O}(S_{q^e})$ to $\mathcal{O}(eS_q)$ is possible. In fact, the full decryption is rarely needed, and one can instead perform much cheaper zero check. Since $h^v = 1$, we have $c^v = (g^m h^r)^v = (g^v)^m$, which means that $m = 0 \mod u \iff c^v = 1 \mod n$, so one can just check the last equality.

2.2 Protocols

Our protocol stack is a modification and improvement of the protocol stack of Bost et al. [BPTG15]. Due to the space constraints and the fact that the batched algorithms presented later are similar in terms of logic and correctness proofs to the non-batched versions, we only describe basic protocols here in brief. For the full versions and detailed explanation see the Appendix B. We will use at most three different encryption schemes variables in them at the same time, named AHE_1, AHE_2 and AHE_3. The encryption with these AHEs will be correspondingly denoted by $[\cdot]$, $[\![\cdot]\!]$, and $[\![\cdot]\!]$. We will later show how to instantiate them with the encryption schemes we have just presented.

Comparison Protocols. A crucial part in the algorithm stack is taken by integer comparison algorithms. The basic unencrypted integer comparison algorithm we use is due to Damgård [DGK08] with several modifications later added by Veugen [Veu11, Veu12] that we will call SecureCompareRaw. This algorithm compares two plaintext values of l bits held by two different parties separately. The server which holds the secret key sends encrypted bits of his number to client, which then transforms them into set of values that indicate whether the compared numbers differ starting from the specific bit or not. These computations are done homomorphically, and then, after a permutation and multiplicative blinding step, are sent back to the server, which performs a trivial zero-check to determine the half of result of the comparison. Server encrypts the results, sends them to client, and client, after homomorphic unmasking has the encrypted comparison bit. The protocol makes use of just three messages, most of the homomorphic computations are done on the client side using AHE_1, and the result is the AHE_2 encrypted bit. DGK scheme was designed specifically for this protocol and is the first candidate for AHE_1. The values contained in the AHE-encrypted ciphertexts in this comparison protocols are less then $log(3l)$ bit. For practical applications, the DGK plaintext space can be very small. The resulting comparison bit on AHE_2 place can be encrypted with any homomorphic scheme.

The next protocol SecureCompare is comparing encrypted integers of l bits, both provided as a client input, both encrypted by AHE_3. The main procedure calls the SecureCompareRaw algorithm after doing a certain blinding conversion to transform two encrypted integers into two plaintext values such that no party learns any information about the original encrypted values. The algorithm works only if the plaintext size of AHE_3 is bigger than $2^{\sigma+l+1}$ bits, to fit the random seed of $\sigma + l$ bits that is used to disguise the original values. This is, as it will be explained later, in most cases unacceptably big for DGK scheme, so the original works suggest using Paillier as AHE_3. The value returned is encrypted using AHE_2, and is a un-blinded version of SecureCompareRaw output. For more details, see the Appendix B.

Argmax. A top-level protocol from [BPTG15] that we will be improving is the computation of the argmax function called Argmax. It is an essential building block of applications like SVM or Naive Bayes classifiers. Its input is (at most) three secret keys on the server side, and a list of m values encrypted with AHE_3 on client side. The purpose of this protocol is to obtain index $i, 0 < i < m$ that indicates the maximum element in the list. The algorithm is linear in number

of communication rounds, and it is essentially the linear search for maximum value. We first permute the list with Π and assign $\Pi(0)$ element as current maximum, and then securely update this value $m-1$ times using SecureCompareRaw protocol and blinding techniques, following the permutation order. At the same time, server maintains a boolean variable saying which index was associated with the last update. In the end of the protocol, this index is sent to the client and, after being mapped by the inverse permutation, returned as the output.

3 AHE Packing Modes

This section provides a description of the plaintext packing problem, current state of the art, and presents two packing approaches.

3.1 Definition and Related Work

The term PAHE (Packed AHE) was introduced in [JVC18] and it abstracts SIMD addition, SIMD multiplication by a scalar, and between-slot permutation, though in this document we will use term PAHE with permutations being optional. PAHE systems can be instantiated with almost all the modern FHEs, and the particular FHE packing techniques make use of cyclotomic fields, and the plaintext algebra is represented by the virtue of CRT as a product of subrings, as explained in [GHS12]. The permutation functionality is achieved by applying the Frobenius automorphism to the encrypted polynomial value, which corresponds to a specific automorphism composition of CRT-packed plaintext. We shall be looking into the non-polynomial ring based setup to achieve the same goal. FHE systems also accumulate noise with each operation, which can be reduced either by using homomorphic bootstrapping, or, as it is done in Gazelle, with 2PC. On the other hand, PHEs do not accumulate any noise, so a packed PHE scheme could eliminate noise related issues in PAHE applications.

There exist solutions that use AHE in the SIMD way, though in a very limited manner. The most straightforward idea to use one AHE ciphertext per slot. One example of this approach with Paillier is [BPB09]. This gives the desired SIMD properties, but induces huge space overhead — now ciphertext takes $2s \log n$ bits for s slots, while plaintext space is much bigger than needed. The advantages of this approach is the simplicity and the ability to perform all the mentioned operations, including permutations, which are just permutations of ciphertexts. In the rest of the section we assume the setup where we perform packing within a single plaintext space.

Another common solution [NWI+13, SSW09, EVTL12, BK17] is to pack plaintext vectors by separating them with precomputed number of zeroes that depends on the specific circuit. This limits operations to homomorphic addition exclusively. To our best knowledge, there exist no solutions doing SIMD multiplication using Paillier or other additive PHE schemes.

DGK Plaintext Space. The significant advantage of DGK encryption is the ability to pick arbitrary plaintext space. Among all the additive PHEs we reviewed this ability seems to be the most useful one from the perspective of designing the plaintext space with specific properties. Comparing to other schemes, Paillier's plaintexts are in $\mathbb{Z}/n\mathbb{Z}$ for RSA-like composite n, and we could not find a way to effectively embed SIMD-like structure inside it. All the options we came up with were either risking to reveal the factors (by picking a custom n) or being ineffective. None of the Paillier's scheme generalisation seem to provide any useful properties too. The cryptosystem of Damgård-Jurik [DJ01] generalises Paillier and extends plaintext space to $\mathbb{Z}/n^s\mathbb{Z}$, while ciphertext space becomes $\mathbb{Z}/n^{s+1}\mathbb{Z}$. This reduces expansion ratio to one asymptotically, but makes decryption algorithm more involved. Packing a vector of s elements inside DJ looks

quite intuitive, but again we could not make use of it beyond the standard zero-padded packing. Moreover, its big ciphertexts make it quite impractical. Groth's cryptosystem is arguably the closest one to the DGK — it also does the computations in the subgroup of $\mathbb{Z}_n^*$, but importantly it does not allow to publish its size (in our case, u), because this weakens the security assumption.

3.2 Packing Approaches

This section describes two ideas that solve the packing problem. The first one is based on the discrete Fourier transform (DFT), more concretely on the number theoretic transform (NTT). In this model, we think of the input vectors as of polynomials, and the ciphertext operations are performed in the DFT evaluation space. The second approach uses Chinese remainder theorem, where target vector is associated with the list of residues modulo primes chosen beforehand. It is worth mentioning that CRT approach is much more straightforward and simpler to implement, and operations in it do not accumulate noise, unlike in DFT packing.

Description and Limitations. When speaking about plaintext packing modes in this document, we usually mean finding an embedding f of a product of commutative rings $\prod R_i$ into the plaintext space R in such a way that certain operations in R correspond to additions and multiplications in $\prod R_i$. In other words, we want $f : \prod R_i \to P$ to be the ring homomorphism, where $P \subset R$. The task is somewhat similar to the fast multiplication techniques, that are commonly done by homomorphic mapping [Ber01], and indeed both our packing methods are using fast multiplication ideas.

It is important to notice that we can not build a fair ring homomorphism between the ring $S = \prod R_i$ and the ring $R = \mathbb{Z}_{uv}$ in the DGK exponent because of the different characteristics, in case all characteristics of R_i are the same. Then $char(S) = n$, and $char(R) = |R| = u$, but $|S| = m^n$, so given that for homomorphism to exist we need $u \mid n$, and hence $u \leq n$, but that means $|R| < |S|$ since for our parameters $n < m^n$. Because of this limitation, we will be looking at partial solutions. For example, we can allow accumulation of extra information (similar to noise in FHEs), limiting the number of homomorphic operations that can be executed. So the homomorphism property will be only true for some elements of the image. This is the idea behind the first packing solution, the DFT packing. We can also pick R_i of different size and bigger than we need, limiting the scope of our SIMD operations to the cases where we do not observe the single slot value overflow (in R_i), thus making the concern regarding characteristics mismatch irrelevant. The CRT packing mode takes this route.

DFT-based Packing. The DFT packing idea is to make use of the convolution theorem to perform convolution product in the evaluation domain that corresponds to the SIMD multiplication in the coefficient domain. For this, we need to select the parameters similarly to the way it is done in NTT. Assume that m is the real plaintext modulus for a single slot (the one we want to obtain), and we would like to have n slots. Then we select $N = kn + 1$ for some k, such that N is prime and $N > m$. Since N is prime, n is invertible in $\mathbb{Z}/N\mathbb{Z}$, and we can find a generator g of $\mathbb{Z}_N^*$. Then $\omega = g^k$ is a n-th primitive root of unity, and we can use N as NTT modulus with ω. One important distinction is that in our forward NTT we will not perform the reductions modulo N, allowing encoded values to grow.

The next step is packing the NTT transformed evaluation form vector into the bigger plaintext space. Assume that DGK plaintext space is $\mathbb{Z}_u$ for some $u = u_0^n - 1$. If N is less than u_0 (where u_0 is the maximum size of a single slot), we can represent NTT transformed vector $f = NTT(p)$ in base u_0 by the naive embedding ψ that maps $(f_0 \cdots f_{n-1})$ to $\sum u_0^i f_i$. Of course, the inverse

ψ^{-1} can be easily implemented by dividing and taking modulo n times. Notice that unless the sum of any two f_i and g_i is bigger than u_0, addition of $\psi(f)$ and $\psi(g)$ results in $\psi(f+g)$.

Since u is chosen as $u_0^n - 1$, we have that for the ciphertext $f = NTT(a)$, $\psi(f) \cdot u_0 \mod u$ is its encoded rotation (circular right shift) of vector f. By multiplying two ciphertexts $\psi(f)$ and $\psi(g)$ for $g = NTT(b)$ having in mind the rotation property, we will obtain a cyclic convolution of f and g in $\mathbb{Z}/u\mathbb{Z}$ that corresponds to the SIMD multiplication of NTT-encoded plaintexts a and b by the virtue of the convolution theorem. The whole transformation is the following: $\mathbb{Z}_m^n \subset \mathbb{Z}_N^n \xrightarrow{NTT} \mathbb{Z}_{u_0}^n \xrightarrow{\psi} \mathbb{Z}_{u_0^n-1}$, and its inverse is precisely $NTT^{-1} \cdot \psi^{-1}$.

Homomorphic addition of two (ψ encoded) ciphertexts f and g results in the ciphertext of at most $\log \max(\|f\|_{\inf}, \|g\|_{\inf}) + 1$ bits. If $\|g\|_{\inf} < N$ (which is a common use case), and $\|f\|_{\inf} \gg \|g\|_{\inf}$ the operation is relatively cheap. When homomorphically multiplying f by a scalar s we perform a convolution product, so it increases the ciphertext ψ slot value by at most $\log \|s\|_{\inf} + \log n$. The extra $\log n$ comes from the fact each slot in the cyclic convolution product is at most sum of n products $f_i \cdot s_i$. Since usually we are multiplying by the scalar $s \in (\mathbb{Z}/N\mathbb{Z})^n$ (NTT-transformed scalar with reduction mod N), we increase the value by at most $\log N + \log n$. The takeaway is that for any given circuit it is always possible to derive an upper bound on how many bits should be reserved for u_0, and the slot value growth is adequate.

Another question is whether it is possible to perform plaintext permutations or rotations on encrypted data. Without having an impossibility proof, we only present the intuition to answer this question negatively, which is the following one. Any multiplication of the exponent with some scalar s is the multiplication with s interpreted as base u_0 decomposed, and thus it corresponds to the SIMD multiplication. Since any element $s \in \mathbb{Z}/u\mathbb{Z}$ can be represented in base u_0, the multiplication always corresponds to the SIMD one in the plaintext space. In the same way, any addition of our exponent to some $u \in \mathbb{Z}/u\mathbb{Z}$ corresponds to the plaintext SIMD addition of the ciphertext with the inverse NTT evaluation of u. Therefore the only two accessible operations on the exponent we store our plaintext in lead to the SIMD operations that do not permute the slots. If ψ encoded vector is pointwise multiplied by $\omega^{i^n}_{i=0}$, then the corresponding plaintext will be rotated by one position to the right. This is because for each $f_j = \sum_{i=0}^{n-1} a_i w_n^{ij}$ multiplying it by w_n^j modulo N leads to $a_{n-1} + \sum_{i=1}^{n-1} a_{i-1} w_n^{ij}$, as if our a_i would shift right. The issue with this approach is that we can not perform the pointwise ψ encoded vector multiplication in the exponent, as we are limited to convolutions. This is what makes our setup different from the BGV-like FHE setup — we are limited to a much simpler ring structure which is forced by the encryption format, and eventually to less operations.

CRT-based Packing. Another, more straightforward packing mode is based on CRT decomposition and to enable it we need to choose $u = k \prod p_i$ where all p_i are distinct primes and $k \geq 1$. Then, the plaintext vector a can be associated with a number equal to $a_i \mod p_i$ and converted to the CRT domain using the CRT algorithm. The inverse operation is just taking the number modulo all the base primes. Since $\prod \mathbb{Z}_{p_i}$ is a ring, both addition and multiplication in the CRT domain correspond to the SIMD addition and multiplication of plaintexts. Unless there are slot value overflows, SIMD multiplication and addition are correct. To encode negative a_i into the group we just encode $p_i - a_i$, and all the other operations work exactly as expected.

We also could not find a possibility to perform rotations using CRT packing. To simplify a question we will view the simplest case — rotating the encryption of the vector with the only nonzero slot. It is always possible to erase all the slots except for the chosen one using SIMD multiplication with a zero-one mask. Let $B_i = \prod_{k \neq i} p_k$, then CRT encoding of vector with value m on the position i (and zeroes on all the other positions) is $C_i = B_i(B_i^{-1} m \mod p_i)$ where the inverse of B_i is taken modulo p_i. This comes from the CRT algorithm directly. Now we want

to change the value in the exponent of DGK encryption from C_i to C_j. One approach could be trying to multiplicatively cancel B_i and $B_i^{-1} \mod p$, but B_i is a zero divisor in $\mathbb{Z}_u$ for any k. We also tried investigating other CRT representations (CRT encoding is only unique modulo $\prod p_i$, but there are many representatives modulo $k \prod p_i$), but without success, and they are also hard to maintain with respect to homomorphic operations. The overall impossibility intuition is quite the same as with DFT packing — we are only limited to the SIMD addition and multiplication, and we can not generalise these operations to permute any input vector.

3.3 Expanding DGK Plaintext Space

Using DGK with a nonstandard small plaintext space raises concerns about its security and performance, which we review now.

The assumption behind DGK scheme is that it is hard to distinguish the group $G = \langle g \rangle$ and $H = \langle h \rangle$, with $H \leq G$, given the two random elements from each one. More formally: (n, g, h, u, x) and (n, g, h, u, y) should be computationally indistinguishable, where n, g, h, u are generated by the DGK key generation procedure, x is uniform in G and y is uniform in H. The concrete DGK instantiation in [DGK08] uses small prime u of size 17 to 37, and v of bit length equal to 160.

The important question to ask is whether picking $\mathbb{Z}_u$ for the particular u as a plaintext space breaks the DGK assumption. Veugen in [Veu12] suggests using u of size 32 bits, which is already much bigger than suggested originally. The short list of possible simple attacks described in [DGK08] (including factoring n or guessing v) does not depend on the size of u, while v is big enough. Another naive attack that checks whether $x^u = 1$ does not seem to get more dangerous when u increases, since the number of witnesses of this kind is in the order of u itself, which is much less than uv possible random values chosen (given that $\log v = 160$). The attack from [DGK09] also does not apply to our scenario.

There are other related concerns and potential attacks described. Groth's work mentions analysis of Naccache and Stern cryptosystem [NS98] which states that publishing a common parameter $\sigma \mid (p-1)(q-1)$ with $|\sigma| > |n|/4$ ($|n|$ here denotes size of n in bits) leads to factorisation attack. Though, NS cryptosystem has slightly different key generation setup, more close to the one of Groth. It also mentions that similar factorisation attack is possible on scheme 3 in the original Paillier cryptosystem, and the subsequent variant [PP99] fixes that by using the similar (to Groth's PKC) subgroup setup, without revealing subgroup size. The σ we pick is exactly u^2, so we need $|\log u| < |n|/8$, which is satisfied in our implementation and practical applications.

4 The Protocol Stack

This section describes the protocol stack that makes use of the PAHE encryption presented earlier. We will present parallel protocols that reduce both the amount of communication and number of homomorphic operations executed. We will denote the PAHE schemes used as $\mathsf{PAHE}_i, i \in \{1 \ldots 3\}$ and keep the previous bracket notation. First several protocols (comparisons and linear batched argmax) are quite similar to the protocols they are based on, so the proofs of correctness are deferred to Appendix B.

4.1 Batched Comparisons

The idea of running the argmax protocol in batches starts from the DGK comparison as it is the most primitive protocol in the chain. Let us show that DGK protocol can be run in parallel,

producing the batch of comparisons as a result, with exactly the same communication complexity and overall communication pattern.

Batched SecureCompareRaw. The high-level intuition is that we can replace any linear function evaluation or any conditional choice by the batched alternative, so since the protocol uses only these operations, it can be parallelized. As we see from Protocol 1, most of the steps of ParSecureCompareRaw resemble steps of SecureCompareRaw, though now most of the values are vectors of k elements, and conditionals of the original protocol are replaced with SIMD multiplications on plaintext masks.

Protocol 1. ParSecureCompareRaw (parallel comparison of unencrypted values)

Server's (S) Input: $(c_0, \dots, c_{k-1})$ with $0 \le c_i < 2^l$; SK_{PAHE_1} and SK_{PAHE_2} that may be the same. $c_{i,j}$ denotes j-th bit of c_i.
Client's (C) Input: $(r_0, \dots, r_{k-1})$ with $0 \le r_i < 2^l$.
Client's Output: $[\![\boldsymbol{\epsilon}]\!]$, where $\boldsymbol{\epsilon}_i = r_i \le c_i$.
The protocol:

1. S sends encrypted bits $\{[\boldsymbol{cb}_i]\}_{i=0}^{l-1}$ to C, where each $\boldsymbol{cb}_i$ is PAHE-encrypted vector of length k consisting of ith bits of $\{c_i\}_{i=0}^{k-1}$.
2. For each $i, 0 \le i < l$ C computes $[\boldsymbol{cb}_i \oplus \boldsymbol{rb}_i]$ with XOR acting per-slot. First C computes $\boldsymbol{y}_i = \{\boldsymbol{y}_{i,j} = r_{j,i}\}$. Then C sets $[\boldsymbol{cb}_i \oplus \boldsymbol{rb}_i] \leftarrow [\boldsymbol{cb}_i \cdot \boldsymbol{y} + (\boldsymbol{1} - \boldsymbol{cb}_i) \cdot \overline{\boldsymbol{y}}]$ where $\overline{\boldsymbol{y}}$ is a binary negation of mask $\boldsymbol{y}$, and vector products are SIMD.
3. C picks a random $\boldsymbol{s}$ of length k with $s_i \in \{1, -1\}$.
4. For each $i, 0 \le i < l$ C computes
   $[\boldsymbol{e}_i] = [\boldsymbol{s} + \boldsymbol{rb}_i - \boldsymbol{cb}_i + \boldsymbol{3}\sum_{j=i+1}^{l-1}(\boldsymbol{cb}_j \oplus \boldsymbol{r}_j)]$
   $[\boldsymbol{e}_l] = [\boldsymbol{s} - \boldsymbol{1} + \boldsymbol{3}\sum_{j=0}^{l-1}(\boldsymbol{cb}_j \oplus \boldsymbol{rb}_j)]$
5. C performs parallel multiplicative blinding of the $l+1$ value computed:
   $[\boldsymbol{e}'_i] \leftarrow \mathsf{ParMulBlind}([\boldsymbol{e}_i])$
   and sends $\{[\boldsymbol{e}'_i]\}_{i=0}^{l}$ to S after permuting them.
6. S generates a bit vector $\boldsymbol{\delta}$ such that $\boldsymbol{\delta}_i = 1$ if any of $\boldsymbol{e}_{j,i} = 0$ for $0 < j \le l$, and $\boldsymbol{\delta}_i = 0$ otherwise. S sends $[\![\boldsymbol{\delta}]\!]$ to C.
7. C sets $[\![\boldsymbol{\epsilon}]\!]$ to $[\![\boldsymbol{\delta} \cdot \boldsymbol{s} + (\boldsymbol{1} - \boldsymbol{\delta}) \cdot \overline{\boldsymbol{s}}]\!]$.

We now discuss several details and points distinct from SecureCompareRaw that are more difficult to spot.

Horizontal Shuffling. The proper way to permute the elements at step 5 is to generate permutation per input row, that affects the particular slots, not only exchanging the $[\boldsymbol{e}'_i]$ sent. This can be done on the client side in the following way. First, we generate k bitmasks $m_i = \{m_{i,j} = (i = j)\}$ to single out every element of every $\boldsymbol{e}'_i$. Then for every slot j, $0 \le j < k$ we compute $[\boldsymbol{v_j}_{,i}] = [\boldsymbol{e}'_i \cdot m_j]$. Each $\boldsymbol{v_j}$ represents the original set of $l+1$ vectors $\boldsymbol{e}'_i$ but with only slot j being left nonzero. Then we permute each $l+1$ values of $\boldsymbol{v_j}$ using different permutation and recombine them all by SIMD adding all $\boldsymbol{v_j}_{,i}$ for j, thus obtaining $l+1$ values where each row was shuffled independently. This operation requires km multiplications and additions, which is a relatively high cost, and the only step of the algorithm. Though, as we will see later, it does not present a huge performance threat on the parameters that we use.

Without this per-slot permutation, the server can learn the approximate frequencies of most significant different bits of client inputs' in this batch, even though their actual positions are

mixed. In order to prove that one top-level permutation is enough to achieve privacy towards S, we would need to show that for any input we could simulate C. But this is not possible, however, and we seemingly can not achieve even statistical privacy. And even in the case when the raw comparison is called from the SecureCompare protocol, which previously additively blinds the values and reduces them modulo 2^l. Here is the short counterexample. Assume that the $\Delta_i = |x_i - y_i|$ is small, for instance $\Delta_i = 1$. Then, even after we blind and compare $r \mod 2^l$ with $(r + 2^l + y_i - x_i) \mod 2^l$, the reduction will yield an absolute difference of Δ_i with high probability when l is big (since $2^l \gg \Delta_i$, we will rarely observe the bigger of y_i, x_i to overflow while the smaller does not). This means that the zeroes (most significant different bits) computed in batched ParSecureCompareRaw will be, using single permutation, distributed within one i, $0 \leq i < l + 1$. The hiding factor s does change this distribution, as it only decreases the number of zeroes observed in about half (for each comparison result, but still in half in total). Hence, it is hard to simulate the client without knowing the input values.

Packing Mode Choice. As it was suggested previously, DGK encryption can be used as a basis for PAHE_1 by using plaintext packing methods. Among the two suggested methods — DFT and CRT packing — the second one is much more appropriate for the argmax protocol suite and ParSecureCompareRaw in particular. It is conceptually simpler, does not accumulate noise, allows to do multiplicative blinding, and can be used to test slot values for being zero as it is done in the original DGK scheme (without performing the real decryption). Let us elaborate on the last two advantages.

Multiplicative Blinding. The crucial part of the DGK comparison is multiplicative blinding, which is performed by the client right after it computes the values e_i that it will then send to the server (ParMulBlind, step 5). Multiplicative blinding essentially leaves the value inside the ciphertext as zero if it is zero, and otherwise distributes it evenly among all non-zero values. The question is how to achieve this in the batched setting. For multiple-ciphertext setting of Paillier we can perform blinding by just SIMD multiplying the value by scalar big enough, which will then be computationally hard to invert. We now inspect how DGK packing modes handle this issue.

If CRT encoding is used — the plaintext space size u is $\prod p_i$ — raising the ciphertext to s_i will preserve the CRT encoding, because if $c = m_i \mod p_i$, then for $c' = c \cdot s_i \mod \prod p_i$ we still observe $c' = 0 \mod p_i$ if $m_i = 0$. And it is easy to see that the blinding property is achieved — if we pick s_i big enough, $c' \mod p_i$ will be distributed evenly (unless $m_i = 0$ or $s_i = 0 \mod p_i$).

This logic does not apply in quite the same way to the DFT packing. Recall that any operations that do not break ψ encoding correspond to some SIMD operations modulo N — the DFT prime. SIMD additions do not seem very useful, since no addition can give us the desired property. And each multiplication that preserves ψ bounds in the evaluation domain is a SIMD product in the coefficient domain. So it is possible to SIMD multiply by the vector of random elements v_i such that its evaluation domain representation has $\sigma + l$ bits. Assume that we blind the vector value a with the mask v. Right after the decryption is done, recovering the multiples a_i and v_i that give the precise coefficient domain representation of $c = a \cdot v \mod N$ is hard because of the statistical masking property. Next, notice that it is possible to do the inverse DFT without modulo N reduction, but this does not help to recover the factors as well. The polynomial c produced in this way has property $c_i = a_i \cdot v_i \mod N$, but $\forall a_i, a_i(a_i^{-1}c_i) = c_i$ so we do not get enough data from the modular equation to restore the factors. The fact that $c_i = n^{-1} \sum_j (d_j w^{-ij})$ is not reduced modulo N seems to be also useless, as its absolute value does not reveal any information about factors because of the size of v. Therefore, the biggest

issue with multiplicative blinding in DFT packing is that it requires to have free $\sigma + l$ bits per slot, which could be quite restrictive.

Fast Zero Checks. The other distinction between the packing modes is the ability to test whether the certain slot of the CRT encrypted value is zero. Assume that m is a packed version of the list of values $\{m_i\}$ and we want to check whether $m_j = 0$. In the original non-packed DGK encryption scheme this check is straightforward. Recall that $h^v = 1$, so $c^v = (g^m h^r)^v = (g^v)^m$. Now $m = 0 \mod u$ (u is the DGK plaintext modulus) iff $c^v = 1 \mod n$. This logic can be easily transferred to a CRT packed plaintext message. Now for j-th slot $m_j = 0$ means that $m = 0 \mod p_j$. Let $B_j = \prod_{i \neq j} p_i$, then $c^{vB_j} = (g^v)^{mB_j}$. Since $m = kp_j$ for some k, $mB_j = k \prod p_i = 0 \mod \prod p_i$. So, again, $c^{vB_j} = 1 \mod n$. Since the values vB_j can be precomputed, the check is just one modular exponentiation and one comparison with 1.

We could not find a way to effectively perform this check for the DFT packing with the setup introduced previously. When a single slot value of a polynomial is equal to zero modulo m, the evaluation representation of the polynomial is still very diverse and does not seem to have any properties that may be used to perform the check. Even when the coefficient-value polynomial is equal to zero modulo N (meaning all its coefficients are zeroes modulo N), the check is quite non-trivial. In this case, polynomial evaluation at any root of unity is zero modulo N, as well as a base u_0 encoded list of its evaluations on all the DFT defined roots of unity. So if N is a zero divisor in $\mathbb{Z}_u = \mathbb{Z}_{u_0^n - 1}$, and $N\hat{N} = 0 \mod u_0^n - 1$, then the encoded value will be 0 mod $u_0^n - 1$ after multiplied by $\hat{N}$. This approach only works if we pick u_0 and n in a way that N is zero divisor. More generally, again, every ciphertext multiplication corresponds to some SIMD plaintext multiplications, and since there are multiple representatives of every plaintext values, we cannot just SIMD multiply and compare, and for any other operations the plaintext subgroup structure looks quite restrictive.

Batched SecureCompare. Now we would like to move one protocol layer up to the *encrypted* values comparison, and present the batched version of SecureCompare. The protocol description is almost syntactically identical to the original one, and hides most of the parallelization into the vector notation, introducing the same small number of technical differences like conditionals already mentioned later.

Protocol 2. ParSecureCompare (batched comparison of encrypted values)

Server's (S) Input: SK_{PAHE_1}, SK_{PAHE_2}, and SK_{PAHE_3}, which may be the same.
Client's (C) Input: $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$, $0 \leq \boldsymbol{x}_i, \boldsymbol{y}_i < 2^l$ for $0 \leq i < k$ both vectors encrypted with SK_{PAHE_3}.
Client's Output: $[\![\boldsymbol{\epsilon}]\!]$, where $\boldsymbol{\epsilon}_i = \boldsymbol{x}_i \leq \boldsymbol{y}_i$
The protocol:

1. C picks random $\boldsymbol{\rho}$ where each $\boldsymbol{\rho}_i$ is of $\sigma + l + 1$ bits, sets $\boldsymbol{r}$ such that $\boldsymbol{r}_i = \boldsymbol{\rho}_i \mod 2^l$, and computes $[\![\boldsymbol{z}]\!] = [\![\boldsymbol{2^l} + \boldsymbol{y} - \boldsymbol{x} + \boldsymbol{\rho}]\!]$. C sends $[\![\boldsymbol{z}]\!]$ to S.
2. S decrypts $[\![\boldsymbol{z}]\!]$, and sets $\boldsymbol{c} = \{\boldsymbol{z}_i \mod 2^l\}_{i=0}^{k-1}$
3. S and C run ParSecureCompareRaw protocol using SK_{PAHE_1}, SK_{PAHE_2} to compare $\boldsymbol{c}$ and $\boldsymbol{r}$. C obtains $[\![\boldsymbol{\epsilon}]\!]$ with $\boldsymbol{\epsilon}_i = (\boldsymbol{r}_i \leq \boldsymbol{c}_i)$
4. S computes $\boldsymbol{zm}$ with $\boldsymbol{zm}_i = \boldsymbol{z}_i / 2^l$ and sends $[\![\boldsymbol{zm}]\!]$ to C.
5. C computes $\boldsymbol{rm}$ with $\boldsymbol{rm}_i = \boldsymbol{r}_i / 2^l$ and returns $[\![(\boldsymbol{x} \leq \boldsymbol{y})]\!] = [\![\boldsymbol{zm} + \boldsymbol{1} - (\boldsymbol{rm} + \boldsymbol{\epsilon})]\!]$.

As in the non-batched protocol, the blinding step in the SecureCompare protocol assumes that the plaintext size of the single slot of PAHE_3 is at least $\sigma + l + 1$ bits. For the encryption schemes available it leaves us with Paillier only. The plaintext size of GM is obviously incompatible, and the concerns about DGK scheme's plaintext size were discussed in the previous section. We have also investigated a question of using DGK for PAHE_3 and converting it to Paillier using protocol, but this does not seem to be possible because of incompatible moduli, and otherwise is reduced to the multiparty modulo computation, which is a nontrivial task [GMS10]. A solution to this may be the perfect security comparison algorithm from [Veu12], which we did not investigate in detail.

4.2 Batched Argmax

Since we now have the protocol that compares two lists of encrypted values, it seems reasonable to generalise the Argmax protocol to produce the k argmax results with the same communication pattern. The algorithm, again, is very similar to its basic variant, and the only non-trivial difference is in the way we update the index values and compute conditional-dependent values. The protocol ParArgmax is presented below.

Per-row permutations in ParArgmax are used in the same way as per-slot permutations in the secure compare, but here they hide a statistical data about relations of maximum updates — the changes of $\boldsymbol{imax}$ on the server side. Since the best candidate for PAHE_1 is multi-ciphertext Paillier, this step is cheap enough, as we only permute complete ciphertexts without changing their internals.

Protocol 3. ParArgmax

Server's (S) Input: SK_{PAHE_1}, SK_{PAHE_2}, and SK_{PAHE_3}. All PAHE_i support at least k slots.
Client's (C) Input: $(\llbracket \boldsymbol{v}_0 \rrbracket, \ldots, \llbracket \boldsymbol{v}_{m-1} \rrbracket)$ with $0 \le \boldsymbol{v}_{i,j} < 2^l$ for $0 \le i < m$, $0 \le j < k$.
Output: Index value $\boldsymbol{imax}$, $0 < \boldsymbol{imax}_j < m$, such that $\boldsymbol{v}_{\boldsymbol{imax}_j}$ is the maximum value among the m values $\boldsymbol{v}_{i,j}$, $0 \le i < m$.
The protocol:

1. C generates a set of permutations $\{\Pi_i\}_{i=0}^{k-1}$ over $\{0 \ldots m-1\}$, and permutes input vectors $\llbracket \boldsymbol{v} \rrbracket$ such that Π_i is applied to m values in slot i only. The resulting list of encrypted vectors is denoted by $(\llbracket \boldsymbol{x}_0 \rrbracket, \ldots, \llbracket \boldsymbol{x}_{m-1} \rrbracket)$.
   C sets $\llbracket \boldsymbol{max} \rrbracket \leftarrow \llbracket \boldsymbol{x}_0 \rrbracket$, $i \leftarrow 1$.
2. S sets $\boldsymbol{imax} \leftarrow \boldsymbol{0}$.
3. Run ParSecureCompare protocol on values $\llbracket \boldsymbol{max} \rrbracket$ and $\llbracket \boldsymbol{x}_i \rrbracket$ to obtain $\boldsymbol{b_i} = [\![(\boldsymbol{max} \le \boldsymbol{x}_i)]\!]$.
4. C picks $\boldsymbol{r_i}, \boldsymbol{s_i} \leftarrow [0, 2^{\sigma+l}]^k$, and sets $\llbracket \boldsymbol{m'_i} \rrbracket = \llbracket \boldsymbol{max} + \boldsymbol{r_i} \rrbracket$, $\llbracket \boldsymbol{x'_i} \rrbracket = \llbracket \boldsymbol{x}_i + \boldsymbol{s_i} \rrbracket$.
   C sends $\llbracket \boldsymbol{m'_i} \rrbracket, \llbracket \boldsymbol{x'_i} \rrbracket, [\![\boldsymbol{b}_i]\!]$ to S.
5. S decrypts $\llbracket \boldsymbol{b_i} \rrbracket$ and sets $\llbracket \boldsymbol{v_i} \rrbracket = \llbracket \boldsymbol{x'_i} \cdot \boldsymbol{b_i} + \boldsymbol{m'_i} \cdot \overline{\boldsymbol{b_i}} \rrbracket$, $\boldsymbol{imax} = \boldsymbol{i} \cdot \boldsymbol{b_i} + \boldsymbol{imax} \cdot \overline{\boldsymbol{b_i}}$
   S reencrypts $\boldsymbol{b_i}$ with SK_{PAHE_3} and sends $\llbracket \boldsymbol{v_i} \rrbracket$ and $\llbracket \boldsymbol{b_i} \rrbracket$ to C.
6. C sets $\llbracket \mathbf{max} \rrbracket = \llbracket \boldsymbol{v_i} + (\boldsymbol{b_i} - \boldsymbol{1})\boldsymbol{r_i} - \boldsymbol{b_i}\boldsymbol{s_i} \rrbracket$
7. If $i < m - 1$, $i \leftarrow i + 1$ and go to step 3.
8. S sends $\boldsymbol{imax}$ to C.
9. C returns $\boldsymbol{imax}'$, where $\boldsymbol{imax}'_i = \Pi_i^{-1}(\boldsymbol{imax}'_i)$.

The intuition of this protocol's application value is that since DGK encryption is as fast for batches as for a single value, we save up on raw comparisons and on communication cost, since we have exactly the same communication patterns. We confirm this intuition on practice in the next section.

4.3 Logarithmic Argmax

Now since we can achieve a performance improvement per single argmax input set in a batched setting, can we use the batched comparison to improve a single argmax computation on just one set of values? Our findings provide the positive result, and we now discuss the resulting algorithm called LogArgmax.

The basic idea is to use batching techniques to decrease the number of rounds performed in the argmax algorithm by updating the maximum values as a set. Instead of doing m rounds of communication, we perform $\log m$ rounds but compare possible maximums with batches. Therefore we require $m \leq 2k$, where k is a upper bound of SIMD elements fitting into the chosen PAHE. Without loss of generality, we will present the protocol with a parameter restriction — $k = 2^{k_0}$ and $m = 2k$. We maintain the list of maximum values $\boldsymbol{a_i}$ throughout the protocol, which is initially equal to our original list of argmax inputs. At the beginning of every round, client additively blinds this encrypted list $\boldsymbol{a}$, sends it to the server, and asks it to "cut it in half". Given that the current list $\boldsymbol{a}$ contains 2^t values, server's response is to decrypt all these values, and re-encrypt them separately, splitting into two lists both of size 2^{t-1}, so that the first half goes into the first list, and the second one — into the second list. All the other elements are irrelevant and can be filled with zeroes, but in practice we write our algorithms to ignore them completely when it's possible. After these two lists are sent back to client, and client unblinds them homomorphically, parties can run the batch comparison to obtain a list $\boldsymbol{c}$ of 2^{t-1} values, where $\boldsymbol{c}_i = max(\boldsymbol{a}_i, \boldsymbol{a}_{i+2^{t-1}})$ for $0 \leq i < 2^{t-1}$. It is clear that after $\log m$ rounds the only one element in the list will be left and it will be equal to the global maximum of the original list. At the same time, server observes all the comparison results and thus can maintain a list saying which indices are passing into the next round, so in the end he will be able to say exactly which index is the argmax result value.

We also need to permute elements inside a single ciphertext vector, since otherwise client discloses the particular maximum update pattern to the server, which is a violation of inputs' privacy. To do that when PAHE_3 is Paillier, we just permute ciphertexts. With DGK, in general, it is also possible to perform the permutation via protocol. The simplest solution to permute a vector of length k is to send it additively blinded vector to the server, which then would extract each component and replicate it into every other slot, producing k^2 ciphertexts with only one element nonzero. These can be sent back, unblinded, and combined according to the permutation chosen. In practice, and given than we use Paillier for SK_{PAHE_3} anyway, instead we stick with ciphertext-per-slot permutation, which is quite fast.

To reduce the communication complexity, we can merge some of the protocol steps. The first cut can be done at the permutation step, and each consecutive one can be merged with the step where server updates his local indices variable. Another improvement is comparing twice as much elements as the PAHE fits on the very first step (choosing $m = 2k$ and not $m = k$), so we will put this into the protocol design to maximise the performance increase given by the batched comparison. This explains that algorithm's inputs are two ciphertexts vectors both of size $2k + 1$, and that the permutation size is $2k$, and it acts on two ciphertexts.

We see that on each step both parties transform two inputs of size 2^m into two inputs of size 2^{m-1}. The transformation is the same as in the original argmax, though now we compare the batch of 2^m elements at once. Server replies with the same arguments as before, but pre-split to avoid extra communication. The security argument is the same as in the original argmax — all the values server sees are either blinded as before, or do not reveal any useful information because of the comparison protocol property. Permutation step prevents the analysis of indices updates.

Protocol 4. LogArgmax

Server's (S) Inputs: SK_{PAHE_1}, SK_{PAHE_2}, and SK_{PAHE_3}, SIMD level of each PAHE scheme is 2^k.

Client's (C) Inputs: A list $\boldsymbol{v}$ of 2^{k+1} values encrypted as two ciphertext vectors $[\![\boldsymbol{v}^l]\!]$ and $[\![\boldsymbol{v}^r]\!]$ of 2^k values each such that $0 \leq \boldsymbol{v}_i^l, \boldsymbol{v}_i^r < 2^l$ for $0 \leq i < 2^k$.

Output: Index value $imax$, $0 < imax < 2^{k+1}$, such that $\boldsymbol{v}'_{imax}$ is the maximum value among the 2^{k+1} values in $\boldsymbol{v}$.

The protocol:

1. C picks a permutation Π over $\{0 \dots 2^{k+1}\}$, and permutes $[\![\boldsymbol{v}^l]\!]$ and $[\![\boldsymbol{v}^r]\!]$ as a single list of 2^{k+1} values into $[\![\boldsymbol{x}^l]\!]$ and $[\![\boldsymbol{x}^r]\!]$.
2. S sets $\boldsymbol{imax} \leftarrow \{0 \dots 2^{k+1} - 1\}$.
3. C and S set $m \leftarrow 2^k$.
4. Run ParSecureCompare protocol on values $[\![\boldsymbol{x}^l]\!]$ and $[\![\boldsymbol{x}^r]\!]$ to obtain $\boldsymbol{b} = [\![(\boldsymbol{x}^l \leq \boldsymbol{x}^r)]\!]$.
5. C picks $\boldsymbol{m}^l, \boldsymbol{m}^r \leftarrow [0, 2^{\sigma+l}]^{2^m}$, and sets $[\![\hat{\boldsymbol{x}}^l]\!] = [\![\boldsymbol{x}^l + \boldsymbol{m}^l]\!]$, $[\![\hat{\boldsymbol{x}}^r]\!] = [\![\boldsymbol{x}^r + \boldsymbol{m}^r]\!]$.
   C sends $[\![\hat{\boldsymbol{x}}^l]\!], [\![\hat{\boldsymbol{x}}^r]\!], [\![\boldsymbol{b}]\!]$ to S.
6. S decrypts (or performs zero-checks of) $[\![\boldsymbol{b}]\!]$ and sets $[\![\boldsymbol{d}]\!] = [\![\hat{\boldsymbol{x}}^r \cdot \boldsymbol{b} + \hat{\boldsymbol{x}}^l \cdot \overline{\boldsymbol{b}}]\!]$,
   S updates maximum values: $\boldsymbol{imax} = \{\boldsymbol{b} \cdot \boldsymbol{imax}_{2^m+i} + \overline{\boldsymbol{b}} \cdot \boldsymbol{imax}_i\}_{i=0}^{i=2^m-1}$
7. If $m = 0$, go to step 10, otherwise:
   S splits $[\![\boldsymbol{d}]\!]$ into $[\![\boldsymbol{d}^l]\!] = [\![\boldsymbol{d}_i]\!]_{i=0}^{2^{m-1}-1}$ and $[\![\boldsymbol{d}^r]\!] = [\![\boldsymbol{d}_i]\!]_{i=2^{m-1}}^{2^m-1}$, and $[\![\boldsymbol{b}]\!]$ into $[\![\boldsymbol{b}^l]\!]$ and $[\![\boldsymbol{b}^r]\!]$ in the same way.
   S sends sends $[\![\boldsymbol{b}^l]\!], [\![\boldsymbol{b}^r]\!], [\![\boldsymbol{d}^l]\!], [\![\boldsymbol{d}^r]\!]$ to C.
8. C sets:
   $[\![\boldsymbol{x}^l]\!] \leftarrow [\![\boldsymbol{d}^l + (\boldsymbol{b}^l - \boldsymbol{1})\boldsymbol{m}^{rl} - \boldsymbol{b}^l\boldsymbol{m}^{ll}]\!]$
   $[\![\boldsymbol{x}^r]\!] \leftarrow [\![\boldsymbol{d}^r + (\boldsymbol{b}^r - \boldsymbol{1})\boldsymbol{m}^{rr} - \boldsymbol{b}^r\boldsymbol{m}^{lr}]\!]$
   Where $\boldsymbol{m}^{xl}$ and $\boldsymbol{m}^{xl}$ are first and second 2^{m-1} elements of $\boldsymbol{m}^x$, $x \in \{l, r\}$.
9. $m \leftarrow m - 1$, go to step 4.
10. S sends $\boldsymbol{imax}_0$ to C.
11. C maps $\boldsymbol{imax}_0$ with Π^{-1} and returns the result.

4.4 Further Improvements

The LogArgmax protocol does not make use of all the slots when comparing the batches on step $m > 1$. The following improvement takes advantage of this situation. Assume we compare values $a_0 \dots a_3$ using two vectors of two elements in each. The result is δ such that $\delta_0 = (a_0 \leq a_2)$, $\delta_1 = (a_1 \leq a_3)$. Putting extra 4 values into the original batch can help us determine the global maximum of all a_i in a single comparison. These four comparisons are (a_0, a_1), (a_0, a_3), (a_2, a_1) and (a_2, a_3). Using this technique we can further reduce, for example, 6 elements (adding a_4 and a_5) to 1 using 4 comparisons mentioned and 8 more. For this we need to compare each of possible maximums of $\{a_4, a_5\}$ and $\{a_0, a_1, a_2, a_4\}$, which results in exactly 8 comparisons. Overall, if $k = 16$ and we compare 32 elements, instead of 5 steps LogArgmax takes we could do just three: on the second step we compare by 8 elements as usual, and inserting two packs of 4 extra, obtaining 6 maximums instead of 8, and the last step compares two packs of 3, inserting extra $4 + 8$ to obtain one single maximum, using 15 comparisons.

It is also possible to transform the unblinding procedure to support this kind of logic. For $\{a_0, a_1\}, \{a_2, a_3\}$ masked with $\{m_0, m_1\}, \{m_2, m_3\}$, server will return $[\![\boldsymbol{d}]\!]$ where $\boldsymbol{d}_0$ contains the maximum of (masked) a_i, at index j, and $\{[\![\boldsymbol{\delta_i}]\!]\}_{i=0}^3$ such that $\boldsymbol{\delta}_{\boldsymbol{i},0} = 1$ only for $i = j$, otherwise it's zero. Client now can compute $[\![\boldsymbol{d} - \sum_{i=0}^3 \boldsymbol{\delta_i} \cdot \boldsymbol{m_i}]\!]$, where $\boldsymbol{m}_{\boldsymbol{i},0} = m_i$, $\boldsymbol{m}_{\boldsymbol{i},j} = 0$ for $j > 0$. After this computation exactly the maximum value will get unmasked. This easily generalises to the cases where more values are compared using extra relations and prefix comparisons are

fixed as usual, and it is also easy to adapt the cutting technique. It is not clear how introduced extra comparisons and the cost of merging them on the client side balances with the decreased communication cost.

5 Implementation

The implementation we provide is twofold. We express the basic algebraic primitives, including encryption schemes functionality, in F*, together with specification proofs. We implement PAHE wrappers with key generation procedures in Haskell, using C bindings to the kremlin-extracted F* code. The protocols and benchmarks are also implemented as part of the testing suite written in Haskell. All the code is accessible in HACL* repository under the volhovm_he branch[1].

5.1 Homomorphic Encryption

We provide an implementation of three homomorphic encryption schemed discussed in this paper: GM, Paillier, and DGK. For each of these we formalise keys structure, encryption and decryption functions, and homomorphic functions, together with proofs. Specifications are based on the library expressing most of the required algebraic tools. This module includes definitions and lemmas about divisibility properties, primes, gcd and lcm, modular arithmetic, finite field inverses, units, multiplicative orders, and various related properties. The specification of GM cryptosystem assumes the minimum viable set of properties of Legendre symbol, and then proves the cryptosystem is secure with respect to them. Most of the GM proofs are the modifications on the Legendre symbol properties. Paillier specification is heavier, and it includes the full proof of correctness assuming only two trivial lemmas expressing binomial expansion, and one assuming the uniqueness of factorisation. We prove all the lemmas related to the properties of encoding functions (two types of injectivity), several modifications of Euler theorem (the original Carmichael's theorem is assumed in the algebra module though), residuosity classes lemmas, properties of L functions, etc. We provide lemmas on the relation of encryption and decryption, and prove homomorphic properties. The algebraic background of DGK cryptosystem is simpler since it is based on the multiplicative order properties of secret values. Since decryption in DGK scheme is computing discrete logarithms in the subgroup, we only provide a specification-level proof of the zero-checking functionality instead. In fact, we also implement the decryption algorithm, but its specification-level counterpart is assumed.

The Low* implementation of these schemes is based on long integers arithmetic. Originally, we have adopted the HACL* bignum suite written for RSA PSS library. We had to add a number of more generalised functions that were not used for RSA PSS implementation without proving their functional correctness with respect to the algebraic specification, since implementing the bignum library was not the primary objective of the internship. However, even though it took a decent amount of effort, later it was decided to switch to the OpenSSL implementation of bignum arithmetic because of the mediocre performance of non-optimised naive implementations we have provided. Since F* is compiled to C, it is easy to link our code against the external library. This requires writing a specification-level declarations with bounded conditions, that were in our case just copied from the naively implemented functions (modular operations), and providing a C implementation for these methods. Since our bignums are stored in a slightly different way compared to the OpenSSL bignums, C methods also perform extra allocations and conversions from and to OpenSSL format. However, given that our final goal was to obtain a provably secure reference implementation rather than the optimised one, the performance of the ported bignum

[1] `https://github.com/project-everest/hacl-star/tree/volhovm_he`

methods is more than sufficient, and the conversion overhead seems to vanish on bigger size plaintexts and heavier bignum functions.

All the described schemes' implementations are proven with respect to functional correctness and memory safety assuming the functional correctness of the underlying bignum library. As it was said before, the decryption of DGK is taking subgroup discrete logarithm, so we did not implement the decryption, but it is not required for the application level. Though, for the intermediate testing and comparisons having the full decryption was desired, and it was implemented separately. DGK decryption is implemented partially in C and partially in unverified HACL* and consists of Pohlig-Hellman reduction, CRT, and for the single DLP we use Pollard's ρ that switches to naive iterative solver for the small subgroup size values.

5.2 PAHE and Protocols

The extracted F* and C code was then connected to the haskell application that implements the protocols based on the abstracted PAHE interface.

PAHE Interface. First, we abstract all the cryptographic primitives under one typeclass called PAHE. The typeclass expresses encryption, decryption, and two homomorphic operations done in parallel. Its instances are Paillier and GM (where each slot is a separate ciphertext) and DGK with CRT packing. The interface also captures the equality to zero check, which is instantiated by a special function for DGK instantiation, and by decryption and check for other instantiations.

The key generation procedures, also being the part of the interface, are implemented in haskell with the help of bignum bindings. Methods for Paillier and GM are straightforward and consist of prime generation and, in case of GM, picking the element y that is non-square modulo both p and q. The DGK key generation includes generating g and h with predefined orders uv and v. Since the number of potential element orders grows exponentially with the number of primes forming the CRT subgroup, our multiplicative order check never generates all of them, but rather looks for the element order throwing away primes making up the factorisation one by one until the local minimum is reached. This helps to reduce key generation timings to acceptable numbers, but they are in no way designed to be fast, and thus are not included into the benchmarks.

PAHE interface also has the implementation of permutations, which is implemented for Paillier and GM naturally by permuting the slots, and this method is replaced by a protocol permutation in case of DGK.

Another important performance feature of PAHE interface is the support for partial operations. Every PAHE ciphertext contains a number indicating how many prefix elements are considered, and are nonzero. All the operations on ciphertexts make use of this number — to a much bigger extent in Paillier and GM (which gives a significant increase in performance), and only for zero checks functionality of DGK (since every check can be done separately).

Protocols Implementation. We implement all the algorithms mentioned — ParSecureCompareRaw, ParSecureCompare, ParArgmax, LogArgmax. We note that in case we choose SIMD level of PAHE equal to 1, we obtain the regular versions of the algorithms (some logic is explicitly turned off for the no-SIMD case). The protocols are abstracted over PAHE, and all protocols higher than ParSecureCompare have three keys that may be the same (ParSecureCompareRaw has two). All the algorithms also explicitly receive the number of elements they are working on (that may be smaller than the parallelism level of the PAHE), which then makes use of the partial-PAHE feature described previously. The communication between threads is implemented using ZMQ networking library, which helps us to easily switch the setup of application from local (using in-process sockets), to the real one that uses TCP.

5.3 Benchmarks and Performance

Table 1. Performance of the Low* implementation in microseconds. Schemes are measured with three different composite n bit size, where $\log n = 1024, 2048$ correspond to 80 and 112 bit security level.

	GM_{512}	GM_{1024}	GM_{2048}	$Pail_{512}$	$Pail_{1024}$	$Pail_{2048}$	DGK_{512}	DGK_{1024}	DGK_{2048}
Enc	2.76	5.27	9.85	351.3	2182	15553	52.5	129.5	438.2
Dec	7.8	26.2	64.6	177.1	1096.3	7831.3	106.75	291.69	692.54
Add	1.92	3.33	7.01	3.15	6.99	18.92	2.18	3.21	6.83
Mul	1.12	2.27	4.42	169.9	1100.9	7762.7	8.48	12.71	26.39

Local Benchmarks. Table 1 represents the results of raw methods microbenchmarks, wrapped using C bindings. The measurements were performed on the single core of Intel i5-7200U 2.5 GHz. The DGK plaintext size in these experiments is chosen to be a single prime 193, which is the smallest prime bigger than $64 \cdot 3$, to fit the SecureCompare computations of 64 bit integers. We do again emphasise that the numbers are given only as a reference to compare to the timings of the future protocols. The apparently slow methods, like Paillier encryption and decryption are the direct consequence of the bignum library performance (and the fact Paillier ciphertexts are $2\log(n)$ bits long), and we made sure not to introduce any significant overhead at the implementation level.

Table 2. Performance of PAHE DGK methods per slot in microseconds, with different SIMD levels, ranging by the number of slots used. Column subscript denotes how much slots is used — just one, or all the k.

	Enc_1	Enc_k	Add_1	Add_k	Mul_1	Mul_k	IsZero_1	IsZero_k
DGK_1	146.9	146.9	3.85	3.85	13.03	13.03	68.99	68.99
DGK_8	171.4	26.93	3.92	0.49	31.57	9.54	81.76	79.46
DGK_{16}	188.6	18.4	3.98	0.24	56.00	9.16	100.7	99.9
DGK_{32}	234.6	15.5	3.95	0.12	98.28	9.02	138.7	149.5

Measurements in Table 2 indicate performance of the PAHE wrapped DGK. We use n of 1024 bits, and pick ciphertext space slot size bigger than $64 \cdot 3 = 192$ (as used in the comparison protocol). The IsZero column measures the performance of zero-checking function specific to DGK that is used in ParSecureCompareRaw algorithm. Since all the PAHE instantiations, including DGK, include prefix-only optimisation, we have different timings when using different number of prefix slots. So generally any operation without SIMD is slightly faster than a single slot operation with SIMD, but this slowdown quickly disappears as number of slots used grows. The only operation that does not show a significant dependence on k is homomorphic addition, since it translates directly into the bignum multiplication, with a slight increase induced by the ciphertext prefix length computations we carry with ciphertexts. Both encryption and multiplication use ciphertext packing that depends on the number of elements used, and the overall big overhead source is modular exponentiation, which becomes slower as plaintext space and exponent grow. Because of this, parallel homomorphic addition does not achieve much higher performance than with $k = 1$. Same applies for IsZero function, and the big absolute value of the range is explained by the fact each zero check is done per slot, so the right interval side is k times bigger that the

left one. This is why it actually becomes slightly slower on higher SIMD — not only it is just linear, but some extra overhead for supporting SIMD exists.

Other two PAHE instantiations (Paillier and GM) have linear performance growth over slots, since all the operations are done iteratively over prefix, and their benchmarks are therefore not included.

Protocol Benchmarks. The performance evaluations of networking-dependent protocols are run using two threads on the machine already mentioned, communicating using a relay node in the local network, that serves as a TCP proxy (it implements simple port forwarding). The LAN latency is very small to match the WAN scenario (around 5-8 ms), so we emulate an extra delay with linux utility called netem such that total latency is 40 ms mean. For comparison we also run our algorithm using in-process sockets — in this situation the latency is equal to zero.

Table 3. Average per-comparison time of comparison protocols, in milliseconds. All benchmarks are performed in LAN with mean 40ms delay, unless marked otherwise as "inp" (in-process). Subscripts stay for the PAHE_1 and PAHE_2 used. k indicate SIMD level used, $r \leq k$ indicates number of elements compared.

Experiment setup	$k, r = 1$	$k = 8, r = 1$	$k, r = 8$	$k = 16, r = 1$	$k, r = 16$
$\mathsf{ParSecureCompareRaw}_{DGK,GM}$ inp	22.4	33.1	13.7	45.4	15.3
$\mathsf{ParSecureCompareRaw}_{DGK,DGK}$ inp	23.4	33.9	13.4	46.4	15.2
$\mathsf{ParSecureCompareRaw}_{DGK,GM}$	185	213	40.0	260	29.2
$\mathsf{ParSecureCompareRaw}_{DGK,DGK}$	190	242	45.8	292	29.5
$\mathsf{ParSecureCompareRaw}_{Paillier,GM}$ inp	410	423	419	402	388
$\mathsf{ParSecureCompare}_{DGK,GM}$ inp	27.0	37.4	17.1	50.7	18.8
$\mathsf{ParSecureCompare}_{DGK,DGK}$ inp	27.0	37.8	17.0	52.3	18.7
$\mathsf{ParSecureCompare}_{DGK,GM}$	210	222	48.4	234	34.8
$\mathsf{ParSecureCompare}_{DGK,DGK}$	232	246	58.5	285	35.0

Table 3 presents performance results for the ParSecureCompareRaw and ParSecureCompare.The setup is same — plaintext slot size is bigger than 192, and ranges stand for different number of elements that we compare. All the integers we compare are 64 bits. The PAHE_3 is instantiated with Paillier, all schemes are are using 1024 bit composite number, statistical security parameter is $\lambda = 80$. The inproc versions demonstrate the significant parallelism-supported increase in execution speed, even though the overall decrease in performance with bigger k is noticeable (when $r = 1$). This is the overhead of constructing PAHE ciphertexts, such as, for example, packing, and for DGK being slower with bigger plaintext space. For inproc version of ParSecureCompareRaw on $k, r = 16$ per-slot execution of comparison takes 15.3 milliseconds, which is 1.46 times faster than with $k = 1$, for $k, r = 8$ we observe 1.6 time increase. This parallel advantage coefficient for ParSecureCompare is 1.43 - 1.58, and here we start to notice the slowdown effects of Paillier encryption, which operates on 2048 bit bignums because of the $\mathbb{Z}_{n^2}$ ciphertext space, caused by non-parallel code taking more execution time. The improvement in the LAN setting is even bigger, for instance raw comparison taking 29.2 ms per slot with $k = 16$, while 185 ms is required for $k = 1$, which is a 6.3 times increase in speed.

Regarding the PAHE combinations, we observe that using Paillier as PAHE_1 is the least-performing option, so we do not use it in this way further. The DGK + GM and DGK + DGK combinations perform almost the same when the parallelism level is high, but overall DGK + DGK timings are bigger. Apparently, the ParSecureCompare homomorphic operations

on PAHE_2 become less significant over time giving space to the much heavier PAHE_1 and PAHE_3, so we do not observe the gap anymore. We also mention that both comparison algorithms' performance stabilizes over time with bigger number of bits compared and bigger parallelism level because of quadratic permutation step that we have added to DGK. This is easy to see in inproc measurements, which show better per-slot performance for $k, r = 8$ rather for $k, r = 16$, but in LAN setting the networking delay still makes $k, r = 16$ the fastest scenario.

Table 4. Runtime of argmax protocols. In linear Argmax versions, k indicates the SIMD level, and how many argmax values are computed at once, timings are per slot.

Experiment setup	$m = 8$, ms	$m = 16$, ms	$m = 32$, ms
$\mathsf{ParArgmax}_{DGK,GM}, k = 1$ inp	342	725	1506
$\mathsf{ParArgmax}_{DGK,GM}, k = 8$ inp	177	378	787
$\mathsf{ParArgmax}_{DGK,GM}, k = 16$ inp	181	389	806
$\mathsf{ParArgmax}_{DGK,DGK}, k = 1$ inp	361	745	1540
$\mathsf{ParArgmax}_{DGK,DGK}, k = 8$ inp	177	388	788
$\mathsf{ParArgmax}_{DGK,DGK}, k = 16$ inp	182	390	806
$\mathsf{ParArgmax}_{DGK,GM}, k = 1$	2106	4476	9219
$\mathsf{ParArgmax}_{DGK,GM}, k = 8$	400	863	1795
$\mathsf{ParArgmax}_{DGK,GM}, k = 16$	338	747	1542
$\mathsf{LogArgmax}_{DGK,GM}$ inp	218	464	1016
$\mathsf{LogArgmax}_{DGK,DGK}$ inp	219	463	1057
$\mathsf{LogArgmax}_{DGK,GM}$	1025	1679	2601

The comparison of argmax protocols is given in Table 4. The hardware setup is the same as for comparison protocols. We measure the performance for two PAHE combinations for every type of protocol. First set of experiments dedicated to the ParArgmax protocol measures how much time does it take to perform the single comparison given the parallelism level k. We observe that for inproc experiments the time needed for one round is reduced almost twice (e.g. $m = 16$, 725 ms for $k = 1$ and 389 ms for $k = 16$, which is 1.81 times less) and for LAN setting we observe that parallel versions perform 6 times faster per slot. Regarding the difference between DGK+GM and DGK+DGK combinations, they seem to perform almost the same on inproc experiments, with DGK+GM being slightly faster on small k values, so we did not test DGK+DGK in LAN.

Regarding LogArgmax protocol, we clearly see that DGK+GM setting is 10-15 percent faster, and this difference decreases when m grows. Compared to the parallel argmax in inproc setting we observe that LogArgmax is 1.3-1.4 times faster, and in LAN setting LogArgmax this coefficient grows to 3.5.

6 Conclusion

In this work we have investigated the possible packing methods of additive PHEs, presenting two packing modes — DFT and CRT packing — specialising for the DGK encryption. We modify DGK parameters slightly and argue why this modification does not affect the underlying security assumption. For the practical implementation we only pick CRT packing. We improve secure comparison protocols and argmax protocol to operate in a SIMD way, and also present a logarithmic argmax protocol that uses batches to speed up a single argmax computation. The base of our implementation is a verified primitives library written in F*, that provides many functional and memory-related guarantees. It includes three homomorphic encryption schemes

— Paillier, GM, and DGK. We implement and benchmark all the protocols described in the work, compare and analyse them with different parameters, such as parallelism level and different encryption schemes' combinations. All the parallel protocols, including LogArgmax, show better computational performance comparing to their non-parallel versions, and the gap is even more significant in the WAN-like setting.

References

AAUC18. Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):79, 2018.

ALW16. Majedah Alkharji, Hang Liu, and CUA Washington. Homomorphic encryption algorithms and schemes for secure computations in the cloud. In *Proceedings of 2016 International Conference on Secure Computing and Technology*, 2016.

Ber01. Daniel J Bernstein. Multidigit multiplication for mathematicians. *Advances in Applied Mathematics*, pages 1–19, 2001.

BGV14. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

BK17. Adil Bouti and Jörg Keller. Secure genomic data evaluation in cloud environments. In *2017 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE, 2017.

BPB09. Tiziano Bianchi, Alessandro Piva, and Mauro Barni. On the implementation of the discrete fourier transform in the encrypted domain. *IEEE Transactions on Information Forensics and Security*, 4(1):86–97, 2009.

BPTG15. Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, volume 4324, page 4325, 2015.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

CBL+18. Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

CHK+18. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

DGK08. Ivan Damgard, Martin Geisler, and Mikkel Kroigard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

DGK09. Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. A correction to" efficient and secure comparison for on-line auctions". *IJACT*, 1(4):323–324, 2009.

DJ01. Ivan Damgård and Mads Jurik. A generalisation, a simpli. cation and some applications of paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*, pages 119–136. Springer, 2001.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

EVTL12. Zekeriya Erkin, Thijs Veugen, Tomas Toft, and Reginald L Lagendijk. Generating private recommendations efficiently using homomorphic encryption and data packing. *IEEE transactions on information forensics and security*, 7(3):1053–1066, 2012.

fst. F*: a higher-order effectful language designed for program verification. `https://www.fstar-lang.org/`.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

GBDL+16. Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

GHS12. Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482. Springer, 2012.

GM84. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

GMS10. Jorge Guajardo, Bart Mennink, and Berry Schoenmakers. Modulo reduction for paillier encryptions and application to secure statistical analysis. In *International Conference on Financial Cryptography and Data Security*, pages 375–382. Springer, 2010.

Gro05. Jens Groth. Cryptography in subgroups of $\mathbb{Z}_n^*$. In *Theory of Cryptography Conference*, pages 50–65. Springer, 2005.

hal14. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.

HS. Shai Halevi and Victor Shoup. An implementation of homomorphic encryption. `https://github.com/shaih/HElib/`. Accessed: 2019-08-10.

JVC18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

KSW+18. Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e19, 2018.

LJLA17. Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–631. ACM, 2017.

MRSV19. Eleftheria Makri, Dragos Rotaru, Nigel P Smart, and Frederik Vercauteren. Epic: efficient private image classification (or: learning from the masters). In *Cryptographers' Track at the RSA Conference*, pages 473–492. Springer, 2019.

MZ17. Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

NS98. David Naccache and Jacques Stern. A new public key cryptosystem based on higher residues. In *ACM Conference on Computer and Communications Security*, pages 59–66. Citeseer, 1998.

NWI+13. Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*, pages 334–348. IEEE, 2013.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

PH78. Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

PP99. Pascal Paillier and David Pointcheval. Efficient public-key cryptosystems provably secure against active adversaries. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 165–179. Springer, 1999.

RRK18. Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018.

RWT+18. M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721. ACM, 2018.

SEA19. Microsoft SEAL (release 3.3). `https://github.com/Microsoft/SEAL`, 2019. Microsoft Research, Redmond, WA.

SSW09. Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *International Conference on Information Security and Cryptology*, pages 229–244. Springer, 2009.

SV10. Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.

SV14. Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

Veu11. Thijs Veugen. Comparing encrypted data. *Multimedia Signal Processing Group, Delft University of Technology, The Netherlands, and TNO Information and Communication Technology, Delft, The Netherlands, Tech. Rep*, 2011.

Veu12. Thijs Veugen. Improving the dgk comparison protocol. In *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 49–54. IEEE, 2012.

WGC18. Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: Efficient and private neural network training. *IACR Cryptology ePrint Archive*, 2018:442, 2018.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

ZBPB17. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.

A Partially Homomorphic Encryption Schemes

Goldwasser-Micali AHE. GM encryption introduced in [GM84] is a very simple one-bit additive homomorphic encryption scheme based on the quadratic residuosity problem — it is easy to determine whether the number is square in $\mathbb{Z}_{pq}$ but only if you know the factors. The key generation consists in generating the composite $n = pq$ of RSA-like size, and a number y such that y is non-square both modulo p and q. The calculation of quadratic residuosity is done using Legendre symbol function $\left(\frac{a}{b}\right)$, which is equal to 1 if a is a square, and -1 if it is non-square, and 0 if it is zero. The encryption of $m \in \{0, 1\}$ is done in the following way: we pick random r with $1 < r < n$, and compute $E(m, r) = r^2 \mod n$ if $m = 0$ and $E(m, r) = yr^2 \mod n$ otherwise. The decryption of $c = E(m, r)$ is a single evaluation of Legendre symbol $l = \left(\frac{c}{p}\right)$. It is equal to $\left(\frac{r^2}{p}\right) = \left(\frac{r}{p}\right)^2 = 1$ if $m = 0$ and $\left(\frac{yr^2}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{r}{p}\right)^2 = \left(\frac{a}{p}\right) = -1$ otherwise. So $D(c) = 0$ if $l = 1$ and 1 if it is $l = -1$. The Legendre symbol evaluation is one modular exponentiation $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \mod p$, and it can be sped up by pre-computing $(p-1)/2$ and $p-1$ — the last one is needed to compare the exponent result to -1. It is also possible to increase the performance of Legendre symbol by using quadratic reciprocity rules, but for our application it does not seem to be required, as the decryption is already fast enough. The homomorphic operations of GM are straightforward — the modular product of two ciphertexts is an encryption of xor of the plaintext, so we have addition modulo 2. We also have a trivial multiplication c by a scalar v: if $v = 0 \mod 2$, then $c = 0$, otherwise we return c. In any case, we can re-randomize the result to hide what v was equal to.

Paillier AHE. The cryptosystem by Paillier [Pai99, PP99] is based on the Computational Composite Residuosity assumption, which refers to computing the residuosity class in $\mathbb{Z}_n \times \mathbb{Z}_n^*$, where n is a RSA-like composite prime. Without delving into the details of the scheme's correctness and the particular assumption used, we briefly discuss the computational part of the scheme's algorithms. The plaintext space is $\mathbb{Z}_n$, the ciphertext space is $\mathbb{Z}_{n^2}^*$. Encryption in Paillier is computing two modular exponentiation and one modular multiplication: $E(m,r) = g^m r^n \mod n^2$, where $r \in \mathbb{Z}_n^*$ is picked at random. To the ciphertext $c = E(m,r)$ we need to compute

$$m = \frac{L(c^\lambda \mod n^2)}{L(c^\lambda \mod n^2)} \mod n$$

where $\lambda = \lambda(n) = \mathrm{lcm}(p-1, q-1)$ is a Carmichael's function, and $L(u) = \frac{u-1}{n}$ is a specific function for computing residuosity classes, where division is just regular integer division, because $n|u-1$ for all the inputs which is the result of the Carmichael's theorem ($\forall w \in \mathbb{Z}_{n^2}^*, w^\lambda = 1 \mod n$). The inverse of $L(c^\lambda \mod n^2)$ can be precomputed to speed up the decryption, and with this optimisation it only takes single exponentiation and division to compute $L(c^\lambda \mod n^2)$, and one modular multiplication to compute the product with the inverse.

The homomorphic properties of Paillier system are coming from the plaintext being stored in the exponent. The multiplication of ciphertexts corresponds to the plaintext addition modulo $\mathbb{Z}_n$, and raising the ciphertext into a plaintext power multiplies the internal plaintext by this power.

B Comparison Protocols

Here we present the comparison protocols and the argmax protocol described in [DGK08, Veu11, Veu12, BPTG15].

Unencrypted Values Comparison. The protocol SecureCompareRaw (usually called DGK comparison) compares two unencrypted values that server and client hold separately. It makes use of at most two different AHEs. The first one is AHE_1, used in the main protocol body, and it is denoted by $[\cdot]$. AHE_1 can be any scheme a plaintext size of which is bigger than $3l$. The DGK scheme was particularly designed for these needs, as it has a particularly small plaintext space, and it has fast zero-checks that can be performed without decrypting the ciphertext (step 6). The output of the protocol in the is the bit $\epsilon \in \{0,1\}$ encrypted with the AHE_2, denoted by $[\![\cdot]\!]$. SecureCompareRaw itself does not limit AHE_2 in any way beyond the fact its plaintext space should have at least two elements, but it is preferred to have the flexibility because of how the DGK comparison will be used further.

The multiplicative blinding step is done differently with different encryption schemes. For DGK scheme it can be done by raising the element to a random exponent s_i. This ensures that zero will stay zero, and other nonzero elements will become indistinguishable from a random uniformly distributed values. The zero check can be performed much faster without decryption, which is a specific property of the original DGK scheme.

Both compared integers have l bits denoted by $c = c_{l-1} \dots c_0$ and $r = r_{l-1} \dots r_0$, where zero bit is the least significant one. The value e_i for $i, 0 \le i < l$ is equal to zero only if $\forall j, i < j < l, c_j = r_j$, and $c_i \neq r_i$. The particular result depends on the blinding variable s set on the client side. If $s = 1$ and $c_j = r_j$ for all $i < j < l$, then $e_i = 0$ is achieved when $c_i > r_i$, and when $s = -1$ the condition is flipped to $r_i > c_i$. Because of the factor 3 and the restrictions on the plaintext space size, no other way to obtain $e_i = 0$ is possible. Step 5 ensures that S does not learn anything about particular values of e_i, and about their relative order. The detection of any

Protocol 5. SecureCompareRaw (comparison of unencrypted values)

Server's (S) Input: c with $0 \leq c < 2^l$, SK_{AHE_1} and SK_{AHE_2} that may be the same.
Client's (C) Input: r with $0 \leq r < 2^l$
Client's Output: $[\![\epsilon]\!]$, where $\epsilon = r \leq c$
The protocol:

1. S sends encrypted bits $\{[c_i]\}_{i=0}^{l-1}$ to C.
2. For each $i, 0 \leq i < l$ C computes $[c_i \oplus r_i]$, assigning the value $[c_i]$ to the result if $r_i = 0$ or $[1 - c_i]$ otherwise.
3. C picks a random $s \in \{1, -1\}$.
4. For each $i, 0 \leq i < l$ C computes
   $[e_i] = [s + r_i - c_i + 3\sum_{j=i+1}^{l-1}(c_j \oplus r_j)]$
   And the last value:
   $[e_l] = [s - 1 + 3\sum_{j=0}^{l-1}(c_j \oplus r_j)]$
5. C performs multiplicative blinding of the $l + 1$ numbers: $[e'_i] \leftarrow MulBlind([e_i])$ and sends $\{[\![e'_i]\!]\}_{i=0}^{l}$ to S in random order.
6. S checks whether any of the $[e'_i]$ sent is zero (for example, by fully decrypting the value, or in a more optimised manner). If any of elements is zero, S sets $\delta_S = 1$, and to 0 otherwise. S sends $[\![\delta_S]\!]$ to C.
7. C sets $[\![\epsilon]\!]$ to $[\![\delta_S]\!]$ if $s = 1$ and to $[\![1 - \delta_S]\!]$ otherwise.

zero on the server side detects the inequality (a particular inequality depends on s though), and so the last step computes the comparison bit homomorphically by changing δ_S depending on s correspondingly.

The last element e_l is computed to remove the ambiguity in case of inputs equality and to achieve the perfect security towards S. Without it, in case $r = c$, none of the e_i will be equal to zero independently of choice of s. After introducing e_l, in case all the bits of r and c are equal, the sum of all $c_j \oplus r_j$ is equal to zero, and hence $s - 1$ is zero only if $s = 1$, and it never zero if $s = -1$. In the end of the protocol we will observe $[\![\epsilon]\!] = [\![1]\!]$ if $s = 1$ because server will detect zero in e_l, and $[\![\epsilon]\!] = [\![1 - 0]\!] = [\![1]\!]$ if $s = -1$, so now the case of equality is handled properly.

Protocol 6. SecureCompare (comparison of encrypted values)

Server's (S) Input: SK_{AHE_1}, SK_{AHE_2}, and SK_{AHE_3}, which may be the same.
Client's (C) Input: $[\![x]\!]$ and $[\![y]\!]$, $0 < x, y < 2^l$ both encrypted with SK_{AHE_3}.
Client's Output: $[\![\epsilon]\!]$, where $\epsilon = x \leq y$.
The protocol:

1. C picks random ρ of $\sigma + l + 1$ bits, sets $r = \rho \mod 2^l$, and computes $[\![z]\!] = [\![2^l + y - x + \rho]\!]$. C sends $[\![z]\!]$ to S.
2. S decrypts $[\![z]\!]$, and sets $c = z \mod 2^l$
3. S and C run SecureCompareRaw protocol using SK_{AHE_1}, SK_{AHE_2} to compare c and r. C obtains $[\![\epsilon]\!] = [\![(r \leq c)]\!]$
4. S computes $z/2^l$ and sends $[\![z/2^l]\!]$ to C.
5. C computes $[\![(x \leq y)]\!] = [\![(z/2^l) + 1 - ((r/2^l) + \epsilon)]\!]$.

Encrypted Values Comparison. The SecureCompare protocol calls SecureCompareRaw protocol on the values after additively blinding them. The correctness of this protocol is based on the fact that $(x \leq y)$ is equal to the l-th bit (the most significant one) of $2^l + y - x$. The plaintext

values that are compared within the subroutine are previously blinded by the random mask big enough so that it is statistically indistinguishable from a random uniformly distributed value. This requirement sets the lower bound on the plaintext space size of AHE_3, which must be at least $\sigma + l + 1$ bits big. The last step that client performs produces the correct result because of the following relation:

$$\begin{aligned}(z/2)^l + 1 - ((r/2^l) + \epsilon) &= (z/2)^l - (r/2^l) + (1 - \epsilon) \\ &= (z/2)^l - (r/2^l) + (c < r) \\ &= (2^l + y - x)/2^l\end{aligned}$$

The last equality is a simple division property better explained in [Veu11]. Notice that since the result of the comparison is either zero or one, the computations can be done modulo any $m \geq 2$. This allows us to instantiate AHE_2 with any scheme we are considering in this work.

Protocol 7. Argmax

Server's (S) Input: SK_{AHE_1}, SK_{AHE_2}, and SK_{AHE_3} which may be the same.
Client's (C) Input: $(\llbracket x_0 \rrbracket, \ldots, \llbracket x_{m-1} \rrbracket)$ with $0 \leq x_i < 2^l$.
Output: $\arg\max_i x_i$
The protocol:

1. C picks a permutation Π over $\{0 \ldots m-1\}$, sets $\llbracket max \rrbracket \leftarrow \llbracket x_{\Pi(0)} \rrbracket$, $i \leftarrow 1$.
2. S sets $index \leftarrow 0$.
3. Run SecureCompare protocol on values $\llbracket max \rrbracket$ and $\llbracket x_{\Pi(i)} \rrbracket$ to obtain $b_i = \llbracket (max \leq x_{\Pi(i)}) \rrbracket$.
4. C picks $r_i, s_i \leftarrow [0, 2^{\sigma+l}]$, and sets $\llbracket m'_i \rrbracket = \llbracket max + r_i \rrbracket$, $\llbracket x'_i \rrbracket = \llbracket x_{\Pi(i)} + s_i \rrbracket$.
   C sends $\llbracket m'_i \rrbracket, \llbracket x'_i \rrbracket, \llbracket b_i \rrbracket$ to S.
5. S checks whether $b_i = 1$, and sets $v_i = \llbracket x'_i \rrbracket$ and $index \leftarrow i$, otherwise (if $b_i = 0$) it sets $v_i = \llbracket m'_i \rrbracket$
   S reencrypts b_i with SK_{AHE_3} and sends $\llbracket v_i \rrbracket$ and $\llbracket b_i \rrbracket$ to C.
6. C sets $\llbracket \max \rrbracket = \llbracket v_i + (b_i - 1)r_i - b_i s_i \rrbracket$
7. If $i < m - 1$, $i \leftarrow i + 1$ and go to step 3.
8. S sends $index$ to C.
9. C returns $\Pi^{-1}(index)$

Argmax. The Argmax protocol computes the maximum element step by step iteratively, and it performs exactly m rounds of comparison, blinding, and maximum update. At each step $index$ on server side indicates the last step on which an update of max was performed. This, up to permutation, is exactly the index of the list's maximum. The additive blinding on step 4, which is then being cancelled at step 6, is needed to hide any actual information about elements that server operates with. The need for permutation is dictated by the fact that otherwise the order of $index$ updates that server observes would leak information about their relation.