A verification framework for secure machine learning

Prasad Naldurg, Karthikeyan Bhargavan

A verification framework for secure machine learning

Prasad Naldurg
Inria Paris

Karthikeyan Bhargavan
Inria Paris

ABSTRACT

We propose a programming and verification framework to help developers build distributed software applications using composite homomorphic encryption (and secure multi-party computation) protocols, and implement secure machine learning and classification over private data. With our framework, a developer can prove that the application code is functionally correct, that it correctly composes the various cryptographic schemes it uses, and that it does not accidentally leak any secrets (via side-channels, for example.) Our end-to-end solution results in verified and efficient implementations of state-of-the-art secure privacy-preserving learning and classification techniques.

KEYWORDS

homomorphic encryption, formal verification, secure multiparty communication, secure software

1 INTRODUCTION

Enterprise machine learning applications consume vast amounts of user data to produce useful analytics, which support a wide range of services and products. As more of these applications use dedicated machine learning services on public cloud servers, protecting this sensitive and personally identifiable information (PII), already a legal requirement in many countries, is becoming a critical concern. From the point of view of enterprises deploying these services on public clouds, the mathematical models used for learning and inference can also be sensitive and need to be hidden from the servers.

In this context, technologies that keep user data and models private or encrypted, either using hardware protection, or by using cryptographic techniques, can be very useful. A typical scenario is a cloud-based service that answers queries by consulting a model that has been trained on a private database. Users may send private queries asking the server to classify a new input. The server should not learn the model, the query or its result, and the client should not learn anything about the model except for the results to its queries. To begin with, storing data in encrypted form on the servers is attractive as the data at rest is resilient to inadvertent disclosure, active attacks, or malicious insiders. However, processing encrypted data is problematic. There are two families of solutions that seek to tackle this problem.

The first line of research requires the data to be decrypted on the server before processing, and relies on trusted hardware solutions such as secure coprocessors [2] and secure enclaves [14, 17, 21, 24, 26] on the server to guarantee its security. The processing is done in secure memory implemented by hardware and cannot be accessed by another application on the server. The other approach is to build on cryptographic schemes like homomorphic encryption (HE) [5, 11, 15, 18] secure multi-party computation (SMC) [20, 25], garbled circuits [15, 18], and functional encryption (FE) [4], which allow clients and servers to jointly compute functions over encrypted or private data without revealing their inputs to each other, extending the scope of guarantees offered to data at rest by encryption to the operations and results of computation. We focus on the HE scenario to illustrate the need for a formal framework in order to build truly secure software implementations. A typical HE algorithm takes an encrypted input x for program P and produces the encrypted result of applying x on the function encoded by P. Note that our work extends to software for SMC and secure enclaves as well, and will comment on this briefly.

We focus on supervised machine learning with two phases. In *model learning*, where the inputs to the learning algorithm are labeled data values, converted to feature vectors $\vec{x}$, and used to learn a model of weights w of a classifier consisting of say k classes $c_1 \cdots c_k$, given by $C(\vec{x}, w)$. In the classification or *prediction* phase, the label $c_j, 1 \leq j \leq k$ for an unseen feature vector $\vec{y}$ input by a client, is predicted using the classifier C as $c_j = C(\vec{y}, w)$. With machine learning as-a-service, a (third-party) server is presented with a query and is expected to return the appropriate class label prediction to the requesting client as described.

For this article, we highlight the classification phase(similar to [18, 20]) and briefly comment on how we extend it to model learning later. With HE, both the model w and query $\vec{y}$ are encrypted using say a public HE key. The prediction classifier is implemented on the server as the homomorphic evaluation function $Eval(C)$. The result of the prediction, c_j, has to be *declassified* and presented to the client that issued the query. In terms of security guarantees, the client should not learn anything about model $\mathbf{w}$ beyond what it can learn from observing the predicted class of its input, and the server should not learn the value of the input, or its predicted class.

HE schemes that can compute arbitrary functions (called fully HE or FHE) are fairly straightforward to implement, but are prohibitively expensive. Even with the latest implementation of HELib [12], general depth-limited homomorphic computations of interest in machine learning have very large overheads, e.g, with matrix multiplication being over 600K times slower than plaintext computations, which does not make them practical for useful applications. However, HE schemes that are restricted in their functionality, called partial HE schemes (PHE) are more practical, and can perform one type, say add or multiply [13, 22] or a small number of computations, e.g., quadratic functions [3].

One of the earliest practical proposals to use PHE for secure classification is the work by Bost et al [5]. The idea is to compose several PHE schemes, and build a library of multi-party protocols that offers all the basic building blocks needed for many machine learning classification algorithms. This library implements distributed algorithms for computing the XOR, addition, dot products, and polynomials over encrypted inputs, as well as protocols for secure comparison and argmax over encrypted and unencrypted inputs. These building blocks are composed together to implement Naïve Bayes, hyperplane decision, and decision tree classifiers efficiently,

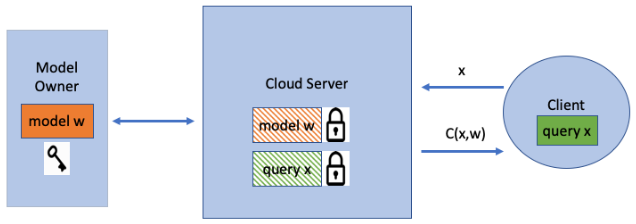

Figure 1: Machine learning classification over encrypted data: Input to the classifier on the server is encrypted user query x, output is class $C(x, w)$. Model w and query x are opaque to the server.

on real world medical dataset examples requiring only milliseconds for classification.

In this and other works that also look at model learning [6, 9, 15, 16, 18–20], several PHE and SMC schemes are composed to compute the classifier of interest, and intermediate results are often masked, declassified and transformed from one scheme to another before they are finally revealed.

As we show next, the security of the composition needs to be analyzed carefully, as different PHE functions have different representations and security assumptions. In addition, the inherent implementation *complexity* of the compositions, reasoning about the *correctness* of the computed results, and the need for rigorous arguments regarding the *security guarantees* in terms of leakages, all point towards a more formal treatment of software development in this domain.

2 SECURE ML PROTOCOLS

We present an example machine learning classifier to illustrate the complexity of protocol implementation in this domain. Our operating environment is described in Figure 1. A typical client presents an encrypted query $x = \vec{x_i}$ to a classifier C implemented on a third-party cloud server as shown. For example, a hospital that presents the encrypted features from a scrubbed record on behalf of a patient, to a cancer-screening classifier hosted on the cloud server, and expects an encrypted response indicating the class of the sample. The output of the classifier (given to the client) is the class c that x belongs to, according to the model w. The idea is that the client does not learn this model w, beyond the output class value c for its input, and the server does not learn anything about the client's input $\vec{x}$ or the model w.

What is implicit in this scenario is who owns the (data used to compute) model w. Model w may not be secret, eg., as in PATE [23], where a public model with enough fidelity that cannot be linked to any user record is generated(privacy-preserving). In this case only the client query needs to be kept secret from the server. If the model is secret (say because it is proprietary), then the decryption key rests with the owner of the model. Both clients and third-party servers only work on encrypted data and have public keys. Even if the client and the model owner belong to the same trust domain knowledge of model parameters could leak information about the training data to the client beyond what can be learnt from query responses.

The problem of model learning on encrypted data, is also an active area of research [6, 9, 16, 18, 20]; and we plan to explore this in future work. The process of learning the model requires additional trust assumptions regarding the provenance and privacy of the labeled samples. Usually, labeled data is sanitized to remove identifying attributes, and we assume that there is one stakeholder who can claim ownership of scrubbed data (similar to model owner in Figure 1). This owner (or a consortium of owners) has control over the declassification of the model parameters. In order to protect privacy, additional filters, including differential privacy mechanisms may be needed for secure declassification. For now, we also do not consider private data leaks that are inherent to a specific machine learning system, such as model inversion attacks [10], which are related but orthogonal to our work. We will explore the integration of security and privacy properties in future work.

2.1 A classifier over encrypted data

Since FHE techniques are largely impractical, researchers have explored the use of PHE techniques along with masking and declassification of partial results to implement faster algorithms and protocols to work directly on encrypted data. One of the first along these lines is the work by Bost et al. [5]. We use their example of a typical Naïve Bayes classifier for input $\vec{x}$ (size d) to illustrate the complexity of a typical protocol in this domain.

Algorithm 1: Naïve Bayes Classifier

```
1 , Input  : $\{p(C = c_i)\}$, $k$ classes, $\{p(X_j = v|C = c_i)\}$
    Output : $k^*$
2 begin
3     $k^* = \mathbf{argmax}(p(C = c_i) \cdot \Pi_{j=1}^{d}(p(X_j = x_j|C = c_i)))$
```

Algorithm 1 presents a high-level specification of the standard Naïve Bayes classifier on plaintext models. The two entities are the server and the client: the server has the classifier and the client the query. Given an input feature vector $\vec{x} = x_1, \cdots, x_d$, capturing an observation, we estimate the conditional posterior probability that its class is c_i, $1 \leq i \leq k$, given by $p(C = c_i, X_1 = x_1, \cdots, X_d = x_d)$ using Bayes' rule as the dot product of the prior and conditional probabilities $p(C = c_i) \cdot \Pi_{j=1}^{d}(p(X_j = x_j|C = c_i))$. The class chosen by the classifier is the one with the highest such probability, represented by index k^*, the argmax of the computed values. (The denominator in Bayes' rule $p(X = x)$ is omitted as it is effectively constant).

The Secure Naïve Bayes protocol for two parties using is shown in Algorithm 2. This example is slightly different from the third-party server model we have discussed so far. Here, the server (S) is trusted to know the model w and the PHE secret key. The client (C) only knows the public keys. The prior and conditional probabilities are represented as logarithms and the dot product operation can now be carried out as addition, suggesting the choice of an additive

Algorithm 2: Secure Naïve Bayes Classifier

```
Input  : S: $(PK_P, SK_P)$, $(PK_{QR}, SK_{QR})$, $w = (P, T_{i,j}(x_j))$
Input  : C: $PK_P$, $PK_{QR}$, $\vec{x} = (x_1, \cdots, x_d) \in \mathbb{Z}^d$
Output : $i_0$ to Client such that $p(\vec{x}, c_{i_0})$
1 begin
2     S creates and sends tables $[\![P]\!]$ and $[\![T_{i,j}]\!]$ using $PK_P$ to C
3     C computes $[\![p_i]\!] = [\![P(i)]\!] \cdot \Pi_{j=1}^{d}([\![T_{i,j}(x_j)]\!])$
4     C and S together run argmax, C gets $i_0 = \mathbf{argmax}(p_i)$
```

PHE scheme such as Pailler [22] and the corresponding PHE keys (PK_P, SK_P).

In this protocol the trusted server encrypts the model and gives it to the client who computes encrypted $[\![p_i]\!]$s, a list of posterior probability values. However, the client cannot directly decrypt these values to get argmax as required. If the client sends the $[\![p_i]\!]$ s directly to the server, then the server learns the magnitudes of the clients features. Instead, the client and server work together using several sub protocols, which include techniques such as permutations, masking, declassification, and re-encryption. First, the client chooses a random permutation over its ciphertexts and picks the first two ciphertexts, so that even if the server learns the which of the two is larger(without learning the magnitude), it will not learn the right index. It then works with the server to implement a protocol to jointly discover which of the two are larger. This protocol internally uses the SecureCompare [7, 8] protocol that works a different bitwise additive homomorphic encryption scheme (quadratic residues or QR [13] as shown), whose secret key is known only to the server. This subprotocol reveals which ciphertext is larger among the pair to the server, without revealing its magnitude.

The client now masks both values compared (which remain encrypted) by adding large random numbers homomorphically and sends the pair to the server. The server decrypts the masked pair and sets the appropriate value as **max** (plus the implicit masking) and updates its **maxindex**. Note that the server does not know the real max value, as it has been masked by the client, or the real maxindex as it has been permuted by the client. This max is re-encrypted and sent back to the client, who removes the masking by subtracting it (homomorphically, without learning the magnitude), and repeats the protocol by picking the next element in the list and the current max, and using SecureCompare again, and moving on to the next encrypted value in the list and so on, until the maxindex is updated consistently all along the list. At the end of the protocol the server sends this maxindex (unencrypted) to the client, who undoes the permutation it picked to obtain the real argmax.

2.2 Security analysis

Already we see that a simple one-line plaintext protocol, when implemented using PHE as presented, requires the orchestration of embedded sub protocols, two different additive homomorphic schemes, additive masking (Paillier), bit encryption, bit-counting operations (GM), declassification, permutations, and re-encryption.

The usual trust assumption in these cryptographic solutions is the *honest-but-curious adversary*, who has access to encrypted data (without access to the secret keys to be able to decrypt it) and follows the computation steps correctly. If one of the participants deviates from the protocol, either through malice or because of a bug, the security guarantees of the algorithm are lost.

We now discuss informally how the algorithm presented above can leak information about the client's inputs. Even without knowing the values, the server can count how many times it updated its maxindex. This is a server who is honest-and-curious, without needing to mount an active attack. In the worst case, if all values in the encrypted list have the same magnitude, the server will learn this information. Whether this is ultimately useful or not will depend on the classifier and features. While this particular vulnerability about counting the number of changes to maxindex can be addressed by a preprocessing step in the encrypted world using homomorphic operations that makes all values unique, without changing the real order, there are other subtle vulnerabilities that can be exploited including side channels in the cryptographic implementations that can reveal some operand values.

We cannot rule out that the server is malicious and mount active attacks eg., in an extended three-party version of this protocol, where the secret keys are held by the data owner. For example, the cloud server can set all weights to identity values, and respond to queries to learn features adaptively. In such a three-party setting, the data owner who has the secret keys will need to be present online to declassify masked/partial results, so rewriting this protocol with a separate model owner and an untrusted public server will require the explicit consideration of this requirement.

The complexity of the construction along with these trust assumptions further emphasizes the need for a formal proof that the implementation of a secure machine learning scheme is correct. Things can go wrong in implementations. The protocol specifications in Bost et al [5] contain minor typos, with the client being given the QR secret keys, as well as the wrong masking value for re-encryption, which will invalidate any security guarantees if implemented as such. The challenge is to show this rewriting will not compromise the security of the computations.

With this is mind, we propose to build a software framework that can be used to implement a range of distributed applications on secret data, such as machine-learning inference protocols using PHE with strong verified guarantees. We would like to maintain these guarantees against both passive and active adversaries, and even if the adversary were able to measure side-channels such as timing leaks. Further, we seek to build and verify practical and efficient implementations, and hence our proofs will account for the low-level optimizations in such solutions.

3 OUR FRAMEWORK

Given a high-level algorithmic specification of a machine learning algorithm on secret data, along with a set of confidentiality constraints on its inputs, our goal is to build and verify its implementation as an efficient distributed (two or three-party) cryptographic protocol. To this end, we design a programming and verification framework based around the F* language [1], as depicted in Figure 2.

F* is a functional programming language with a type system that includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus [1]. The language

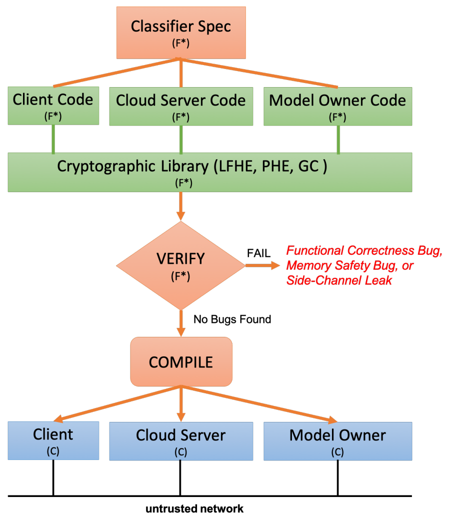

Figure 2: Programming and Verification Framework: Programmer writes high-level mathematical specification of the classifier (or any other computation over private data) in F*. The programmer can run and test this specification, implement this specification as a distributed program with components running at the client, model owner, and cloud server. The program is composed with a cryptographic library and the whole system is verified using F*. If verification succeeds, the code is compiled to C and can be deployed on the network.

is aimed at program verification, and its type system allows the expression of precise and compact functional correctness and security property specifications for programs, which can be mechanically verified, with the help of an SMT solver. After verification, and F* program can be compiled to OCaml, F#, C, or even WebAssembly, and can run in a variety of platforms.

We propose to develop our verified applications in four stages, as shown in Figure 2:

(1) **Global High-Level Specification**: We first write a global high-level specification of our desired distributed computation in F*, focusing on classification algorithms for now. The specification consists of the function ϕ it computes, the characterization of its model w, in terms of feature vectors $\vec{\chi}$, input $\vec{x} \in \vec{\chi}$, and the result $c_i = \phi(w, \vec{x})$ from C the set of classes. The high-level confidentiality specification is that the evaluation of ϕ must preserve the secrecy of w, $\vec{x}$, and c_i from different parties. (We are deliberately generic with the notation; the particular domains, ranges and categories need to be clearly articulated for each algorithm.)

(2) **Distributed Implementation**: We then write implementations, also in F*, of the three parties, detailing all their network interactions and cryptographic computations. We prove that this implementation meets the high-level spec, while preserving our desired confidentiality goals, given an abstract (trusted) interface for the underlying cryptography. The implementation can itself be broken into a reusable verified library of commonly used constructions, like addition, secure comparison, dot products, polynomial evaluation, etc. and application-specific code for the classification algorithm we seek to implement.

(3) **Cryptographic Instantiation**: The code for the three parties will usually rely on a variety of cryptographic primitives, which will need to be instantiated with concrete schemes such as Paillier, GCs, random permutations, etc. which are themselves hard to implement correctly. We build verified implementations of all the cryptographic schemes we need, as an extension to the HACL* verified crypto library [27]. HACL* is written in a subset of F*, but is compiled to C code that is as fast as state-of-the-art hand-written crypto libraries. Each primitive is verified for memory safety, resistance to common timing side-channels, and functional correctness with respect to a high-level mathematical specification. Our plan is to build a library of verified HE and 2PC schemes in HACL*, which will also be reusable in other applications.

(4) **Low-Level Executable Components**: Finally, we compile all our F* code along with the cryptographic library to C to obtain three C libraries, one for each party. We envisage that these libraries will be embedded into larger applications that will handle less security-critical concerns like user interfaces, networking code, and persistent storage. Generating C code allows our code to run efficiently on a variety of platforms, including smartphones, and enables existing legacy applications to use our toolchain to verify their core cryptographic components.

At the end of this workflow, what we have is high performance verified protocol code in C for Clients, Servers and Model Owners, which can communicate over an untrusted network, but still provide strong correctness and confidentiality guarantees.

We have implemented F* specifications of the secure Na"ive Bayes classifier from Bost et al., [5] and built the underlying implementations of Pailler and GM PHE schemes. These implementations are verified against confidentiality, correctness, and side channel resilience requirements and compile down to efficient C libraries. With our implementation we are able to show the vulnerability discussed by declassification in SecureCompare, and provide a fix, which is verified within the framework. We also show how to implement secure permutations and techniques for converting inputs to unique values, which are of general utility to protocols in this domain.

4 FUTURE DIRECTIONS

Our grand goal is to build a comprehensive framework, which will have all the libraries to implement efficient verified protocols for privacy preserving machine learning, in addition to the security and functional correctness requirements we discussed. We plan to extend our F* specifications to include garbled circuits, SMC, and linear algebra primitives at both levels 2 and 3 in our framework and derive optimized implementations in level 4, which can be incorporated into large applications and provide strong guarantees. At the specification level, we plan to work on abstractions to implement and verify Convolutional Neural Networks (CNNs) [18, 20] for image classification and propose to integrate privacy-preserving mechanisms into our tool kit as well.

REFERENCES

[1] [n.d.]. F*: A Higher-Order Effectful Language Designed for Program Verification. https://www.fstar-lang.org/. Accessed: 2018-11-01.

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. 2013. Orthogonal security with cipherbase. In *Proc. of the 6th CIDR, Asilomar, CA.*

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

[4] Dan Boneh, Amit Sahai, and Brent Waters. 2011. Functional Encryption: Definitions and Challenges. In *Theory of Cryptography*, Yuval Ishai (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–273.

[5] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.*

[6] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2017. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. Cryptology ePrint Archive, Report 2017/1114.

[7] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. 2007. Efficient and Secure Comparison for On-Line Auctions. In *Information Security and Privacy*, Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson (Eds.).

[8] Ivan Damgard, Martin Geisler, and Mikkel Kroigard. 2008. A Correction to &#39;Efficient and Secure Comparison for on&#45;Line Auctions&#39;. *Int. J. Appl. Cryptol.* 1, 4 (aug 2008).

[9] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16).* 201–210.

[10] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).* 1322–1333.

[11] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme.* Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.

[12] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-homomorphic Encryption Scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11).* Springer-Verlag, Berlin, Heidelberg, 129–148. http://dl.acm.org/citation.cfm?id=2008684.2008697

[13] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic Encryption &Amp; How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC '82).*

[14] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. 2017. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *CoRR* abs/1711.02279 (2017).

[15] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. 2017. Pretzel: Email Encryption and Provider-supplied Functions Are Compatible. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17).* 169–182.

[16] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Catherine Jones. 2017. Privacy-preserving Machine Learning in Cloud. In *Proceedings of the 2017 Cloud Computing Security Workshop (CCSW '17).* 39–43.

[17] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. *CoRR* abs/1803.05961 (2018).

[18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[19] Manish Kesarwani, Akshar Kaul, Prasad Naldurg, Sikhar Patranabis, Gagandeep Singh, Sameep Mehta, and Debdeep Mukhopadhyay. 2018. Efficient Secure k-Nearest Neighbours over Encrypted Data. In Proceedings of EDBT. (2018).

[20] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. 2019. EPIC: Efficient Private Image Classification (or: Learning from the Masters). CT-RSA 2019.

[21] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16).*

[22] Pascal Paillier. 1999. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes.* Springer Berlin Heidelberg, Berlin, Heidelberg, 223–238. https://doi.org/10.1007/3-540-48910-X_16

[23] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Úlfar Erlingsson. 2018. Scalable Private Learning with PATE. Proceedings of the 6th International Conference on Learning Representations ICLR 2018. (2018).

[24] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15).* 38–54.

[25] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: Efficient and Private Neural Network Training. In *Privacy Enhancing Technologies Symposium.* (PETS 2019).

[26] Yan Zhai, Lichao Yin, Jeffrey Chase, Thomas Ristenpart, and Michael Swift. 2016. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16).*

[27] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).* 1789–1806.