SANDIA REPORT
SAND2020-9254
Printed September 3 2020

Sandia National Laboratories

Securing machine learning models

Kenneth Goss, Benjamin Jackson, Jacek Skryzalin

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

We discuss the challenges and approaches to securing numeric computation against adversaries who may want to discover hidden parameters or values used by the algorithm. We discuss techniques that are both cryptographic and non-cryptographic in nature. Cryptographic solutions are either not yet algorithmically feasible or currently require more computational resources than are reasonable to have in a deployed setting. Non-cryptographic solutions may be computationally faster, but these cannot stop a determined adversary. For one such non-cryptographic solution, mixed Boolean arithmetic, we suggest a number of improvements that may protect the obfuscated calculation against current automated deobfuscation methods.

ACKNOWLEDGMENT

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

DRM digital rights management

GES graded encoding scheme(s)

IoT Internet of Things

IP intellectual property

MBA mixed Boolean arithmetic

MBP matrix branching program

LWE learning with errors

ML machine learning

1. INTRODUCTION

1.1. The goal of securing computation

As the usage of machine learning (ML) increases in high-consequence national security applications, it will become increasingly important to secure trained ML models against adversaries who may wish to reverse engineer or exploit these models. As a first line of defense against such adversaries, one may hope to keep models secure by limiting access to the models (e.g., by deploying models only on a secure network) and by limiting the number of real-world observable effects which result as direct consequences of a model's prediction (so as to hinder an adversary's black-box reverse engineering efforts). At the same time, current trends see the mass deployment of ML models in a variety of environments not controlled by the owner of the ML model (e.g., IoT and other "smart" devices, cloud computing servers).

This report discusses various partial solutions to the problem of securing ML models (or, more generally, any arithmetic circuit) on untrusted platforms. We desire three properties from our security solution:

- **Correctness.** A secured model must produce an output for any legitimate input, and this output should be approximately equal to the output which the raw, unsecured model produces upon receiving the legitimate input.
- **Security.** It should be difficult for an adversary to deduce the model's parameters. We are of the opinion that the description of the model, by itself, is not sensitive; rather, it is only when a model is combined with its parameters that the model becomes sensitive. Furthermore, because of the correctness condition, we assume that an adversary is able to obtain outputs for chosen inputs. Thus, our security condition is that the adversary is not able to learn much more about the model than what is deducible from these inputs and outputs.
- **Computational complexity.** Finally, our model must be somewhat lightweight. That is, evaluation of the model must not place a substantial burden on the computational platform.

We wish to briefly address and provide an illustrative example of the third property above. One potential use case for our scheme is to secure classifiers, deployed by an email service provider, intended to identify spam email or malicious email attachments. One possibility is for the content to be classified to be encrypted and sent to some third party cloud platform with an analytics engine. In this scenario, the email service provider wishes to hide their algorithm's parameters (and possibly the email to be classified) from the cloud computing platform. The cost (which incorporates both monetary aspects and runtime) of such a security scheme includes internet communication with the cloud platform and paying for cloud resources as the cloud performs its

classification. If this cost to the email service provider is higher than the cost to the provider of standing up its own analytic engine, then the provider will choose not to use the security solution.

Unfortunately, solutions that provide the most security, which are usually cryptographic in nature, are often computationally infeasible. On the other hand, less secure techniques, the most notable of which being software obfuscation, can be bypassed by a sophisticated adversary, especially if the adversary is aware of the obfuscation techniques used to scramble the model. Even more troublesome is the question of how effective such measures might be when an adversary has black-box access to a ML model. For example, if the model in question is known to be linear regression, an adversary need only observe n input/output pairs (where n is the length of the model input) in order to completely reconstruct the parameters of the model. The ease of recovery of other machine learning models is largely an open question, especially in situations where an adversary can control the inputs fed to the model.

We wish to emphasize this point – our goal, and the best we can hope for, is to make the algorithm as much of a black box as possible. Most machine learning models are learnable, almost by definition. This means that given enough inputs and outputs (such as those used to train the algorithm), the adversary can reconstitute the algorithm with high fidelity. Thus, the measures discussed in this report are not "one-stop" solutions; but rather components that could aid in establishing "defense in depth".

In the national security setting, we assume a somewhat powerful adversary, who is able to examine the contents of any file stored on the system. Such an adversarial model is not unrealistic, even in everyday applications. In the cloud computing setting, it is entirely reasonable to suspect that a system administrator has full access to all files stored on the system. In most situations, this ability allows the adversary to (theoretically) view the memory state of the program during execution, as the adversary can run the program with chosen inputs and monitor memory contents during program execution. As such, simply storing an encryption of a model's parameters on disk is not a sufficient solution, since these parameters will be decrypted (and placed into memory) when the model is used.

Once an adversary has reconstructed a model, they can take measures to circumvent or fool the ML classifier. For example, in our email example, if an adversary gains access to model parameters, they can alter a malicious email attachment until it appears benign by the classifier. In other situations, adversaries may use evasion attacks [14] to misclassify an input in order to effect or avoid a certain response from the ML model's owner. In high-consequence national security settings, it is of high importance to avoid such situations.

1.2. Decision functions of common machine learning classifiers

Machine learning encompasses a variety of paradigms, including classification, regression, dimension reduction, manifold learning, clustering, and anomaly detection. During single-class classification, one takes an input, often given as a vector $\boldsymbol{v} \in \mathbb{R}^n$, and outputs the category or label to which the input is thought to belong. There are multiple aspects to each classification

procedure, including the underlying model, how the model is trained given the (labeled) training data, and how one uses the model to assign a category to the test data (which should not be processed during training).

In this section, we discuss the final aspect mentioned above. That is, we discuss the various decision functions (i.e., underlying mathematical functions that are used to assign a category to data) used in machine learning classification algorithms. It is these decision functions that we hope to secure. Given an input $\boldsymbol{x}$ and model parameters $\boldsymbol{\theta}$, we let $f(\boldsymbol{x};\boldsymbol{\theta})$ denote the decision function of the machine learning classifier.

It is often the case that multiple classification algorithms can share the same decision functions. However, the way that $\boldsymbol{\theta}$ is determined from the training data differs depending on the underlying model. Although it is not our goal to discuss exactly how these differences arise, we wish to provide an illustrative example of how these differences may arise.

For our example, assume the existence of two categories – red and blue. Both linear support vector machines and linear discriminant analysis have decision functions of the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}\cdot\boldsymbol{x} + b.$$

It is our goal to set $\boldsymbol{\theta}$ so that $f(\boldsymbol{x};\boldsymbol{\theta}) > 0$ for red data and $f(\boldsymbol{x};\boldsymbol{\theta}) < 0$ for blue data. When using linear discriminant analysis, $\boldsymbol{\theta}$ is set so that the hyperplane $f(\boldsymbol{x};\boldsymbol{\theta}) = 0$ is equidistant from the arithmetic mean of the red data and the arithmetic mean of the blue data. However, when using a linear support vector machine, $\boldsymbol{\theta}$ is set so that the hyperplane $f(\boldsymbol{x};\boldsymbol{\theta}) = 0$ separates the red and blue data that are “most difficult" to separate.

We now provide an exposition of the most common decision functions used in machine learning classifiers.

1.2.1. Ensemble methods

It is common when using certain algorithms to create *ensembles* of classifiers. The philosophy of an ensemble is that we may average the results of multiple mediocre classifiers in order to create a powerful classifier. We will introduce ensembles of classifiers in this section, leaving exposition of individual base classifiers for later sections.

There are three primary properties of a base classifier that make it amenable for use in an ensemble: simplicity, diversity, and power.

Simplicity. First, because the runtime of an ensemble is equal to the product of the number of components of the ensemble and the runtime of the base classifier, it is common to use base classifiers whose decision function is easy to compute. Classifiers whose decision functions have high runtime are generally not suitable for use in an ensemble.

Diversity. Second, we observe a greater benefit to using an ensemble when the (trained) base classifiers are sufficiently diverse such that they contribute different pieces of information to the overall ensemble. For example, due to how linear support vector machines are typically trained, one expects the parameters $\boldsymbol{\theta}$ of a linear support vector machine to be roughly identical each time

a linear support vector machine is trained on data sampled from a fixed distribution. As such, an ensemble of linear support vector machines is not expected to perform substantially better than an individual support vector machine.

Power. Finally, we need each individual classifier in the ensemble to perform better than random. Empirically, we observe that it is not necessary for the individual classifiers to perform *much* better than random, because the averaging performed by the ensemble can significantly boost classification accuracy. In fact, using an ensemble with simpler base classifier often performs as good as, if not better than, an ensemble with more complex base classifiers (which are more prone to overfitting to the training data).

1.2.2. Polynomial models

Linear models. A linear model has a decision function of the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}\cdot\boldsymbol{x}+b.$$

Linear models are perhaps the simplest of the machine learning classifiers. As such, they are the easiest to train, most robust to noisy data, and easiest to analyze theoretically. However, the simplicity of the model comes at the cost of reduced expressive power; linear models are traditionally only useful when the data categories are quite distinct or if the data dimension is very high.

Classifiers with an underlying linear decision function include classifiers built from least squares (ridge classification, lasso classification, etc.), linear discriminant analysis, linear support vector classification, and logistic regression.

Higher-order models. In order to extend the expressibility of linear models, one can replace each raw data vector $\boldsymbol{x}$ with a vector $\boldsymbol{z}$, each entry of which is a product of entries in $\boldsymbol{x}$. The decision function for these models is of the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}\cdot\boldsymbol{z}+b.$$

Because the decision function is computed via a dot product of model parameters and input data, these classifiers are still linear models. However, higher-order models can be more expressive due to the possibility of including higher-order terms (e.g., $\boldsymbol{x}_1^2\boldsymbol{x}_4$) in the vector $\boldsymbol{z}$.

Models that use decision functions of this form include some forms of support vector classifiers (although these are used much less frequently than linear and kernel support vector classifiers) and quadratic discriminant analysis. Quadratic discriminant analysis has the a decision function $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}\cdot\boldsymbol{z}+b$, where $\boldsymbol{z}$ includes all linear and quadratic terms in the entries of $\boldsymbol{x}$.

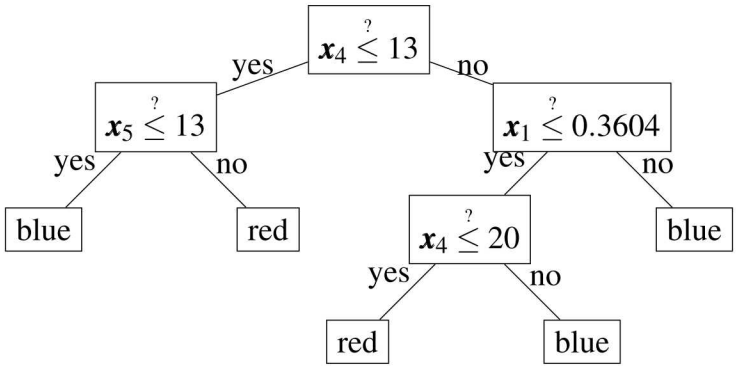

Figure 1-1. A typical decision tree. This particular tree has a depth of 3.

1.2.3. Nearest neighbors classifiers

Nearest neighbor classifiers serve as relatively simple, intuitive, and powerful classification algorithms. The parameters $\boldsymbol{\theta}$ of a nearest neighbor classifier include all of the training data (and their associated labels). Nearest neighbor classifiers operate under the principle that training data close to an input $\boldsymbol{x}$ should have the same label as $\boldsymbol{x}$. Thus, the decision function $f(\boldsymbol{x};\boldsymbol{\theta})$ first finds the k nearest points of training data to the input $\boldsymbol{x}$ and subsequently assigns a label to $\boldsymbol{x}$ based on the labels associated to these points.

Because the decision function associated to a nearest neighbor classifier involves finding the k closest training points to the test point, these classifiers are often used only when the training set is relatively small. Furthermore, nearest neighbor classifiers are typically only used when the dimensionality of the data is small. This heuristic is due to the "curse of dimensionality" – a general principle that states that for data sets encountered "in the wild", as the dimensionality of the data increases, the density of data decreases to such an extent that all data appears equally similar to all other data regardless of category.

1.2.4. Decision trees

Decision trees have a decision function that predicts a label for an input $\boldsymbol{x}$ based on a sequence of tests. These tests are typically of the form $\boldsymbol{x}_i \overset{?}{\leq} c$. Traditionally, the result of each test determines which entry i and threshold c are used in the subsequent test. At the conclusion of a sequence of tests, the decision tree assigns a label to the input datum $\boldsymbol{x}$.

Decision trees are intuitive and perhaps the most easily interpretable of all decision functions used by machine learning algorithms (cf. Figure 1.2.4). Unfortunately, individual decision trees are either prone to overfitting on the training data or otherwise have poor generalization performance.

Thus, decision trees are commonly used in ensembles (cf. Section 1.2.1), where the results of multiple decision trees are averaged to construct a more accurate classification. Decision trees are

particularly suited for ensemble learning due to their simplicity and the diversity of trees provided by most training procedures (which also guarantee that each tree performs better than random).

When used in ensembles, decision trees typically have low depth (typically either 2 or 3). However, it is an intriguing idea that, should one have an system where ensembles are easily implemented but individual classifiers are more difficult to implement, one may be able to create an ensemble of trees of depth 1 (i.e., each tree consists of a single test of the form $\boldsymbol{x}_i \overset{?}{\leq} c$).

Altering the internal tests of decision trees. It may be possible to increase the power of individual trees by replacing each internal node by the decision function of a linear classifier. This idea has not been explored in the literature, both because ensembles of traditional decision trees perform so well and because it is a nontrivial (although not infeasible) task to devise and implement a training procedure for these types of trees. Nevertheless, should one have a system where ensembles are not easily implemented but individual decision trees are easy to implement, one might consider expanding the space of possible tests internal to a decision tree as a means of increasing the classification power of a decision tree.

1.2.5. Kernel support vector machines

Support vector machines (which typically allow the separation of only two classes) operate under the general principle of trying to find a decision boundary that separates only the members of each class that are most difficult to disambiguate. This philosophy contrasts with many other classifiers (e.g., linear and quadratic discriminant analysis), which consider the distribution of *all* members of each class.

To augment the discriminative power of linear support vector machines, it is common for a support vector machine to first embed the data into a space with a high (or even infinite) dimension and subsequently train a linear support vector machine. Due to Lagrangian duality, it is not necessary to compute this higher-dimensional representation; one must only be able to compute the inner product of the result of embedding two of the (lower-dimensional) inputs into a higher-dimensional space.

A function k that allows the computation of this inner product is called a *kernel*. When our input data comes in the form of vectors in $\mathbb{R}^n$, by far the most popular kernel function is the radial basis function: for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, this function is defined as

$$rbf(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\gamma \|\boldsymbol{x} - \boldsymbol{y}\|^2\right),$$

where γ is a hyperparameter that is chosen by the user. Another kernel function, less commonly used but easier to compute, is the polynomial kernel, which has the form:

$$k(\boldsymbol{x}, \boldsymbol{y}) = (\gamma \langle \boldsymbol{x}, \boldsymbol{y} \rangle + b)^d,$$

where γ, b, and d are hyperparameters that are usually chosen by the user.

The decision functions of kernel support vector machines are of the form

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = \sum_i w_i \cdot k(\boldsymbol{x}_i, \boldsymbol{x}) + b.$$

The sum in this decision function is over vectors $\boldsymbol{x}_i$ known as *support vectors*. Each support vector is usually an element of the training data. However, the size of support vectors is usually much smaller than the size of the training set. This set is determined when training the model.

1.2.6. Neural networks

Neural networks form a large class of models, with seemingly countless variations and hyperparameters to set and tune. At their core, neural networks are the result of interweaving layers of linear computation and simple nonlinear functions. Because these simple nonlinear functions are often of the form $\sigma(x) = \max(0,x)$ (the ReLU function) or $\sigma(x) = \frac{e^x}{e^x+1}$ (the sigmoid function), neural networks are not polynomial models. However, there has been research on approximating the nonlinear functions of a neural network with low-degree polynomials (so that the resulting model is polynomial).

For many applications, neural networks are currently the best-performing classifiers. However, this power comes at a cost: neural networks require large amounts of training data, training neural networks can be rather time-consuming, and the underlying workings of neural networks are very poorly understood in general.

We now describe the decision function for multilayer perceptrons (MLPs), which are arguably the most basic class of neural network. Let $\boldsymbol{W}_i$ denote matrices and $\boldsymbol{b}_i$ denote vectors. Furthermore, for a vector $\boldsymbol{v}$ and a function $\sigma : \mathbb{R} \to \mathbb{R}$, let $\sigma(\boldsymbol{v})$ denote the result of applying σ to each coordinate of $\boldsymbol{v}$. When using a MLP for classification, one first associates each category or label to an index j. The decision function $f(\boldsymbol{x};\boldsymbol{\theta})$ of a MLP chooses the category or label of $\boldsymbol{x}$ to be the index of

$$\hat{f}(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{W}_d \cdot \sigma(\cdots \sigma(\boldsymbol{W}_2 \cdot \sigma(\boldsymbol{W}_1 \cdot \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2) \cdots) + \boldsymbol{b}_d$$

that contains the highest value.

1.2.7. Bayesian methods

Bayesian methods impose some probability model on the data and use the underlying model's probability density function as a decision function. For example, if the red data has a probability density function p_{red} and the blue data has a probability density function p_{blue}, then the model will classify an input $\boldsymbol{x}$ if and only if $p_{\text{red}}(\boldsymbol{x}) > p_{\text{blue}}(\boldsymbol{x})$.

Bayesian methods are unique in their ability to incorporate prior knowledge or expertise into the model. Using Bayesian techniques, one can guide the model towards certain parameters of the underlying probability distribution. For example, one can specify that the red data is likely distributed normally with a mean that is roughly equal to $\boldsymbol{\mu}$ and a covariance that is roughly equal to $\boldsymbol{\Sigma}$.

For simple base models, the decision functions of Bayesian methods reduce to standard maximum likelihood estimates, many of which are linear. For example, given parameter vectors $\boldsymbol{\lambda}^{(\text{red})}$ and

$\boldsymbol{\lambda}^{(\text{blue})}$ for a Poisson naive Bayes model, we will predict that a sample $\boldsymbol{x}$ is red if and only if

$$\left(\sum_i \boldsymbol{\lambda}_i^{(\text{blue})} - \sum_i \boldsymbol{\lambda}_i^{(\text{red})}\right) + \left(\sum_i \boldsymbol{x}_i \left(\log \boldsymbol{\lambda}_i^{(\text{red})} - \log \boldsymbol{\lambda}_i^{(\text{blue})}\right)\right) > 0.$$

Note that this decision function is linear.

1.3. Commercially used primitives

In this section, we discuss many useful primitives that each provide partial solutions for computing in untrusted environments, and we identify the inadequacy of these solutions when applied to machine learning models in the presence of a moderately sophisticated adversary.

1.3.1. Digital rights management

The question of how to best secure intellectual property (IP) has long been an active and important research question. Traditionally, this question has been tackled primarily by those studying digital rights management (DRM). DRM is the study of how to protect content, traditionally software and entertainment media, from being pirated. For example, product keys, such as those to be used during installation, can help to prevent piracy, especially when the key must be authenticated with a central server that limits the number of times a particular key can be used. Other systems choose to store media in an encrypted state, with decryption occurring when the media is consumed by the user. For these systems, the decryption key is assumed to be stored securely and inaccessible to the user.

Our problem is similar in that we wish to prevent content from being examined by an adversary. However, due to our adversarial model and operational constraints, the space of solutions is quite smaller than that available to DRM researchers. Because our adversary has the ability to view the memory contents during the execution of the ML model, model parameters must never be present in memory in the clear during execution of the model. Furthermore, any ML model deployed into an operational environment needs to be fully functional, implying that, just like a legitimate user, the adversary should have the ability to interact with the model without the model's parameters ever being stored in the clear. This renders ineffective many common DRM solutions.

1.3.2. Code obfuscation and white-box cryptography

Code obfuscation and white-box cryptography are two techniques that, while often considered DRM techniques, are worth exploring further here. Code obfuscation is the process of replacing source or binary code with functionally equivalent code that is harder to understand. Although many techniques exist for code obfuscation, one often assumes that these techniques can be overcome by a sophisticated adversary. Indeed, many of the tools used to obfuscate code can be run "in reverse" to undo the obfuscation.

White-box cryptography is the study of obfuscating cryptographic algorithms [38, 86]. In particular, one often wishes to decrypt some ciphertext without revealing the decryption key. White-box techniques often achieve this goal by encoding the function that decrypts a ciphertext with a fixed secret key as a series of lookup tables in such a fashion that it is difficult to recover the secret key from these tables. Although white-box techniques provide some degree of security, the security of these schemes is in question due to the large number of attacks on such schemes [57, 70].

Furthermore, white-box techniques generally assume that it is possible to consider only a few bits at a time when performing encryption or decryption (e.g., the S-box of AES operates on 8 bits at a time). This assumption is violated by most ML models. Although it is reasonable to expect that one could decompose addition operations into a set of sub-operations requiring a small number of bits, it is not immediately clear how one would do the same for multiplication operations. Furthermore, most white-box techniques provide no support for nonlinear control flow (e.g., "if" statements) when the control flow is determined at runtime (as might be done when implementing the function $f(x) = e^x$ with a Taylor series).

1.4. Another approach

The remainder of this report will discuss how other tools from cryptography might be used to provide a (partial) solution to our problem. The methods we discuss are generally thought to be of interest to the research community, but have not yet made their way to mainstream use.

2. HOMOMORPHIC ENCRYPTION

2.1. Homomorphic evaluation of ML models

Homomorphic encryption is a technique that allows computation on encrypted data [2, 69]. Many common encryption cryptographic algorithms have some form of homomorphic capability. For example, the Paillier cryptosystem [73] is additively homomorphic (i.e., given encryptions of x and y, one can construct an encryption of $x+y$) and the RSA cryptosystem [78] is multiplicatively homomorphic (i.e., given encryptions of x and y, one can compute an encryption of $x*y$). These limited homomorphic capabilities are sometimes sufficient to compute a function of interest, but solutions in this space, even when available, must be tailored to the specific homomorphic capabilities of the underlying scheme.

There are, however, fully homomorphic cryptosystems which permit arbitrary computation on encrypted values. First introduced by Gentry in 2009 [48], fully homomorphic cryptosystems have become prominent in cryptographic research. A (fully) homomorphic encryption scheme $\texttt{HE}$ consists of the following probabilistic polynomial time algorithms:

- **Key generation.** Given a security parameter λ, $\texttt{HE.KeyGen}(1^\lambda)$ produces a public encryption key $\texttt{pk}$, a secret decryption key $\texttt{sk}$, and an evaluation key $\texttt{evk}$.
- **Encryption.** Given an input m, $\texttt{HE.Enc}(m,\texttt{pk})$ produces a ciphertext $\texttt{ct}_m$ for m.
- **Decryption.** $\texttt{HE.Dec}(\texttt{ct}_m,\texttt{sk})$ produces the plaintext m from an encryption $\texttt{ct}_m$ of m.
- **Evaluation.** Given a function f, the evaluation key $\texttt{evk}$, and a sequence of encryptions $\texttt{ct}_{m_i}$ of messages m_i, $\texttt{HE.Eval}(f,\texttt{evk},\texttt{ct}_{m_1},\ldots,\texttt{ct}_{m_d})$ produces an encryption of $f(m_1,\ldots,m_d)$.

Generally, one assumes that the functions f are restricted to arithmetic expressions consisting of addition, subtraction, and multiplication. Nonetheless, it is possible with just these three operations to emulate any binary circuit due to the universality of and, xor, and not gates (corresponding to multiplication, addition, and addition by 1, respectively).

Fully homomorphic cryptosystems have arrived in three "generations". The first, introduced by Gentry in [48], relies on ideal lattices and is now thought to be less efficient than more recent instantiations. Gentry's approach was simplified in [84], where similarities between the current fully homomorphic encryption schemes were noted.

These similarities gave rise to the second and third generations of fully homomorphic cryptosystems, which are built more directly on learning with errors (LWE) and its ring-based

variant R-LWE [75]. Schemes from the second generation of fully homomorphic cryptosystems (e.g., [11, 26, 27, 29, 43]) follow the general formula

$$\texttt{ciphertext} = \texttt{public key} \times \texttt{random} + \texttt{plaintext}.$$

Addition and multiplication of ciphertexts is allowed due to the fact that the public keys in these schemes can be "cancelled" using the secret key. There have been multiple discoveries which allow one to modify the base schemes from this generation in order to add minor additional functionality. For example, computation on (both floating and fixed) point numbers (rather than integers) is allowed by the techniques of [8], and improvements in efficiency at the expense of some loss in precision are possible using the techniques of [33].

The third generation of fully homomorphic encryption follows the philosophy that a plaintext number x can be encoded as an approximate eigenvalue of a (ciphertext) matrix M with an approximate eigenvector as the secret key [30, 50]. These schemes have been highly optimized (see, e.g., [34, 35, 36, 44]), to the point that it takes roughly 10 ms to homomorphically add or multiply ciphertexts. However, these schemes generally require that the message space be restricted to elements of $\{0, 1\}$.

Although there has been a substantial amount of research in fully homomorphic encryption, the existing schemes remain too computationally expensive for many applications. Nevertheless, there has been a significant amount of research into how homomorphic encryption might be used to secure ML models [9]. This research generally falls into one of two thrusts. In the first, the primary goal is to train a model from encrypted data (e.g., by homomorphically evaluating the optimization algorithms used to train ML models) [55, 62]. In the second, the goal is to encrypt a trained ML model's parameters and to produce an encryption of the decision output by the ML model (which might then be sent back to a user with the decryption key) [60]. Much of the work in this category is focused on homomorphic evaluation of neural networks [24, 31, 37, 51, 56, 87]. In this latter category, [24] is especially interesting due to a trick used to compute (an encryption) of the sign function; this technique is improved in [25] and may be generalizable to other machine learning algorithms.

The second of these thrusts is much more heavily researched, for a number of reasons. First, due to the work that must be done by a data scientist when designing, refining, and testing a model, it is difficult to train a performant model without the ability to view training data. Second, the mathematical operations that must be performed when training a model are usually more complex than those that must be performed when deploying or testing the model, so it is much more difficult to train a model on encrypted data than it is to test a model on encrypted data. Finally, it is often the case that training data, collected by a data scientist for the purpose of training a model, is less sensitive than testing data owned by a user.

The work cited above provides a viable solution when a secret key is available to produce a decision. However, because it is our desire that only an encryption of the model's parameters be used while still obtaining a result in the clear, we would like to be able to produce a decision without requiring a secret key. In all variants of homomorphically encrypted models that we've considered, a key that is used to decrypt the result or decision can be used by an adversary to recover the encrypted parameters of the model.

2.2. Secret keys reveal all or nothing

Decryption in many homomorphic encryption systems is often a dot product $\mathtt{ct}_m \cdot \mathtt{sk}$, for $m \in \mathbb{Z}_t$. For this section, we define $\mathbb{Z}_t = [-t/2, t/2)$ and assume that t is a multiple of 2. For simplicity of exposition, assume that q is a multiple of t; the results presented in this section hold even when q is not a multiple of t, but the exposition is somewhat more clumsy. One might expect that, should we wish to reveal a limited amount of information about m without revealing all of m, we might be able to alter the ciphertext and secret key to values $\tilde{\mathtt{ct}}_m$ and $\tilde{\mathtt{sk}}$ such that, for example,

$$\tilde{\mathtt{ct}}_m \cdot \tilde{\mathtt{sk}} = \begin{cases} 1 & \text{if } m \in [-\frac{t}{4}, \frac{t}{4}] \\ 0 & \text{otherwise} \end{cases}.$$

Our goal would be to publish $\tilde{\mathtt{sk}}$ so that the user can perform such a calculation (where m is the result of the calculation of some decision function). However, we were unable to construct a $\tilde{\mathtt{sk}}$ that doesn't otherwise ruin the security of the underlying encryption scheme.

We provide an illuminating example here, based off the homomorphic encryption scheme by Brakerski [26], modified slightly so that the plaintext space is $\mathbb{Z}_t = [-t/2, t/2)$ rather than $\{0,1\}$ and to incorporate more recent developments regarding the construction of the secret key. For a distribution χ, we denote χ^n the distribution of vectors of dimension n where each entry is sampled from χ. We assume that distributions χ in this section have support in $[-B, B]$. We let $\lfloor x \rceil$ denote the integer closest to x, and $[x]_t$ to be the integer in $\mathbb{Z}_t$ equivalent to x modulo t. The scheme is as follows:

- **Key generation.** Given a security parameter λ, we produce an integer q and a distribution χ over $\mathbb{Z}$ (bounded by $B \ll q$), chosen to provide λ-bit security. The secret key $\mathtt{sk}$ is a vector sampled from χ^n. Let $N = (n+1) \cdot (\log q + O(1))$. Sample A uniformly from $\mathbb{Z}_q^{N \times n}$ and sample $\boldsymbol{e}$ from χ^N. Let $\boldsymbol{b} = [A \cdot \mathtt{sk} + \boldsymbol{e}]_q$, and define the public key as the matrix $\mathtt{pk} = [\boldsymbol{b}; -A]$. The construction of the evaluation key is somewhat more complex; we refer the reader to Section 4 of [26] for more details.
- **Encryption.** Given an input $m \in \mathbb{Z}_t$, $\mathtt{HE.Enc}(m, \mathtt{pk})$ is calculated via

$$\mathtt{ct}_m = \left[\mathtt{pk}^\top \cdot \boldsymbol{r} + \left\lfloor \frac{q}{t} \right\rfloor \cdot \boldsymbol{m} \right]_q \in \mathbb{Z}_q^{n+1},$$

  where $\boldsymbol{r}$ is a vector sampled uniformly from $\{0,1\}^N$ and $\boldsymbol{m}$ is the vector $(m, 0, ..., 0) \in \mathbb{Z}_t^{n+1}$.
- **Decryption.** Given a ciphertext $\mathtt{ct}_m$ and secret key $\mathtt{sk}$, one decrypts m via

$$m = \left[\left\lfloor \frac{t}{q} \cdot [\mathtt{ct}_m \cdot (1, \mathtt{sk})]_q \right\rceil \right]_t.$$

- **Evaluation.** Evaluation is as described in Section 4 of [26].

Although we do not discuss evaluation keys in depth, one relevant detail regarding their construction is that each evaluation key is produced by encrypting the powers of 2 of the bit decomposition of one secret key $\mathtt{sk}$ using a (possibly different) secret key $\mathtt{sk}'$ (this process is performed to reduce the "noise" inherent in homomorphic encryption schemes).

Define $\tilde{\mathtt{sk}} = \left(\frac{2}{t}, \tilde{\tilde{\mathtt{sk}}}\right)$, where $\tilde{\tilde{\mathtt{sk}}}$ will be defined in order to make a quantity discussed later very small. Then

$$\mathtt{ct} \cdot \tilde{\mathtt{sk}} = \boldsymbol{r}^\top \left(\frac{2}{t}(A \cdot \mathtt{sk} + \boldsymbol{e}) - A \cdot \tilde{\tilde{\mathtt{sk}}}\right) + \frac{2}{t}\left\lfloor \frac{q}{t} \right\rfloor m.$$

Thus, for some $I \in \mathbb{Z}$, we have

$$\left[\mathtt{ct} \cdot \tilde{\mathtt{sk}}\right]_q = \boldsymbol{r}^\top \left(\frac{2}{t}(A \cdot \mathtt{sk} + \boldsymbol{e}) - A \cdot \tilde{\tilde{\mathtt{sk}}}\right) + \frac{2}{t}\left\lfloor \frac{q}{t} \right\rfloor m + Iq.$$

Hence,

$$\left\lfloor \frac{t}{q}\left[\mathtt{ct} \cdot \tilde{\mathtt{sk}}\right]_q \right\rfloor = \left\lfloor \boldsymbol{r}^\top \left(A\left(\frac{2}{q}\mathtt{sk} - \frac{t}{q}\tilde{\tilde{\mathtt{sk}}}\right) + \frac{2}{q}\boldsymbol{e}\right) + \frac{2}{q}\left\lfloor \frac{q}{t} \right\rfloor m \right\rfloor + It.$$

Because we are assuming that t is a multiple of 2 and that q is a multiple of t,

$$\left[\left\lfloor \frac{t}{q}\left[\mathtt{ct} \cdot \tilde{\mathtt{sk}}\right]_q \right\rfloor\right]_2 = \left[\left\lfloor \boldsymbol{r}^\top \left(A\left(\frac{2}{q}\mathtt{sk} - \frac{t}{q}\tilde{\tilde{\mathtt{sk}}}\right) + \frac{2}{q}\boldsymbol{e}\right) + \frac{2}{t}m \right\rfloor\right]_2.$$

Let

$$\tilde{e} = \boldsymbol{r}^\top \left(A\left(\frac{2}{q}\mathtt{sk} - \frac{t}{q}\tilde{\tilde{\mathtt{sk}}}\right) + \frac{2}{q}\boldsymbol{e}\right),$$

so that

$$\left[\left\lfloor \frac{t}{q}\left[\mathtt{ct} \cdot \tilde{\mathtt{sk}}\right]_q \right\rfloor\right]_2 = \left[\left\lfloor \tilde{e} + \frac{2}{t}m \right\rfloor\right]_2.$$

It is our desire that this quantity $\left[\left\lfloor \frac{t}{q}\left[\mathtt{ct} \cdot \tilde{\mathtt{sk}}\right]_q \right\rfloor\right]_2$ equal $\left\lfloor \frac{2}{t}m \right\rceil$. This translates into a desire that $\tilde{e}$ be close to zero (say, $|\tilde{e}| \approx \frac{1}{100}$). The size of $\tilde{e}$ impacts the value of $\left[\left\lfloor \tilde{e} + \frac{2}{t}m \right\rceil\right]_2$ when $m \approx \pm t/4$, since this is where small differences in m might change the result of rounding.

We next discuss how to choose parameters so that $\tilde{e}$ is small by discussing bounds on its summands. Note that the value $\left|\frac{2}{q}\boldsymbol{r}^\top \cdot \boldsymbol{e}\right|$ is bounded by $\frac{2NB}{q}$, which, given typical choices of n, q, and B, is usually very close to 0. The more significant contribution to $\tilde{e}$ comes from $\boldsymbol{r}^\top \cdot A\left(\frac{2}{q}\mathtt{sk} - \frac{t}{q}\tilde{\tilde{\mathtt{sk}}}\right)$. We leave as an open question how to construct such variables (including $\tilde{\tilde{\mathtt{sk}}}$) so that the resulting value is very small.

The more concerning aspect of this construction is that, should an adversary be able to compute $\tilde{e} + \frac{2}{t}m$ such that $\left[\left\lfloor \tilde{e} + \frac{2}{t}m \right\rceil\right]_2 \approx \left[\left\lfloor \frac{2}{t}m \right\rceil\right]_2$, then, by multiplying by $\frac{t}{2}$, the adversary will be able to deduce m (within a margin of error of $\frac{t}{2}\tilde{e}$). This alone is sufficiently alarming. However, because of the construction of the evaluation keys (which are known to be encryptions of powers of 2 of the bits of a secret key), it is trivially possible to use $\tilde{\mathtt{sk}}$ to recover many (if not all) of the secret keys.

As such, any value $\tilde{sk}$ that allows a user to deduce whether $m \in [-t/4, t/4]$ that is constructed as described above will also render insecure the entire encryption scheme.

3. FUNCTIONAL ENCRYPTION

Functional encryption is a cryptographic paradigm that, at least theoretically, allows a user to extract $f(x)$ (the result of evaluating a function f on input x) given only an encryption $\texttt{Enc}(x, \texttt{key})$ of x and an evaluation key $\texttt{sk}_f$ for f [21, 22, 72]. Research into functional encryption is traditionally approached from two directions. Following the first approach, which follows a "top-down" philosophy, one builds functional encryption schemes for general functions f. The second approach follows a "bottom-up" philosophy. In these approaches, one uses currently established primitives to construct functional encryption schemes for a greatly restricted class of functions. Furthermore, these approaches are often computationally efficient, but currently only very simple functions can be computed under these schemes.

Functional encryption arose out of developments in identity-based encryption [10, 58, 90]. An identity-based encryption scheme is essentially a public-key cryptosystem where the public key is common knowledge, but where only the holder of a particular identity can perform the decryption operation. Later, these schemes were generalized using techniques from multi-party computation so as to allow a user to a user with identity x to obtain an encryption of a function f and calculate $f(x)$ without revealing anything about f besides $f(x)$ [81]. This is *almost* what we want, but like most techniques used in multi-party computation, its security relies on the security of various parties. In particular, the protocols from [81] rely on the security of a trusted central authority which plays an active role in the encryption process. The goals of functional encryption are to remove the reliance on this trusted third party and much of the communication that is present in [81].

These approaches are primarily focused on the privacy and security of the user input x. Some approaches (e.g., [81] above) are additionally concerned with *functional privacy* – that is, the privacy of the function f (e.g., [20, 83]). In our applications, we assume that the user is controlling the computation, so we do not require privacy of the user's input x. On the other hand, we very much require privacy of the function f. One proposed solution is to switch the roles of the function f and the user input x in established functional privacy schemes. This idea does not immediately provide a viable solution because established schemes typically require a secret key to construct the evaluation key $\texttt{sk}_f$.

3.1. Functional encryption – the "top-down" approach

The "top-down" approach to functional encryption focuses on providing the ability of a user to compute $f(x)$ for general functions f. These solutions frequently assume the existence of cryptographic primitives not known to have secure instantiations, and the estimated computational burden for the calculation of $f(x)$ is several minutes and gigabytes of RAM for

even the simplest of functions [66]. The results of [28] imply that any general functional encryption scheme can be composed with a symmetric encryption scheme to construct a function-private functional encryption scheme.

Furthermore, there are a number of impossibility results that preclude the possibility of various flavors of functional encryption schemes [3, 13, 53]. These impossibility results imply that there must be a secret key shared by the party constructing the evaluation key $\mathtt{sk}_f$ and the party encrypting the input x. These impossibility results are disappointing, but do not immediately apply to our use case because (a) we do not require the privacy of the input x, (b) we do not require a scheme to apply to general functions f, and (c) we wish only to hide the parameters of f (rather than the general form of f).

The approach of Goldwasser et al. provides a way to evaluate functions by composing fully homomorphic encryption, attribute-based encryption (a generalization of identity-based encryption [74]), and Yao's garbled circuits [52]. This approach is improved in [18] using a novel attribute-based encryption scheme with increased efficiency. The techniques of [18] can also be used to perform predicate encryption (a generalization of attribute-based encryption where the "attributes" are private) [54].

There is another approach to functional encryption that relies on *multilinear maps* or *graded encoding schemes* (GES) (see [5] for an overview of graded encoding schemes and their security, and [45, 49, 68] for explicit constructions). Note that although there is a difference between multilinear maps and GES, when in the functional encryption setting, both terms usually refer to GES. Graded encoding schemes are similar to homomorphic encryption schemes; the most significant difference in terms of functionality is that one is sometimes able to test whether a certain ciphertext is an encryption of zero. These primitives are incredibly powerful but are poorly understood, and many have doubts on their security [5]. Furthermore, there are massive runtime and storage requirements of implementations using GES [66]. Graded encoding schemes can be used in two different ways to produce a functional encryption scheme. In the first, one uses the GES to obfuscate a circuit representing a function [6]. In the second, one first represents the function using a matrix branching program and then performs all arithmetic operations using a GES [46].

The second of these approaches is used to (somewhat) concretely describe a scheme for calculating $\mathbb{1}_{[x \leq y]}$ given encryptions of x and y [19]. The same functionality can be implemented using more standard assumptions (e.g., the existence of pseudorandom functions, pseudorandom permutations, and cryptographic hash functions) [32, 67]. The benefit of using the approach of [19] is that it can be composed with other functionality that has been similarity protected by matrix branching programs and graded encoding schemes. The primary detriment of [19] is its draconian computational requirements.

3.2. Functional encryption – the "bottom-up" approach

The "bottom-up" approach to functional encryption focuses on secure and efficient ways for a user to compute $f(x)$ given an evaluation key $\mathtt{sk}_f$ for certain classes of functions f. Most of the

approaches in this category focus on linear functions or subspace membership. For example, the early publication of Shen et al. focus on functional privacy of the functionality $f_v(x) = \mathbb{1}_{[v \cdot x = 0]}$ [83] or $f_M(x) = \mathbb{1}_{[Mx=0]}$ [20]. Most subsequent publications focus on the evaluation of the function $f_v(x) = v \cdot x$ [1, 4, 15, 39, 62]. Many of these also incorporate function privacy, and most utilize pairings and the security of Diffie-Hellman problems.

More recently, schemes have arisen which permit computation of quadratic functions $f_M(x,y) = x^\top My$ [12]. Although the techniques introduced in [12] do not hide the matrix M, they do permit a number of interesting applications. In particular, we can compute predicates using $O\left(n^{2^{d-1}}\right)$ group elements, where n is the bit length of the input and d is the depth of a Boolean circuit computing the predicate. For example, one can construct an M such that $f_M(x) = \mathbb{1}_{[x \leq y]}$ (for some fixed y) using a matrix M whose size is approximately $2\sqrt{N}$ by $1 + \sqrt{N}$, where N is a predetermined upper bound for the values of x and y. Note that, in this example, the y here is easily deduced from M, and is thus completely public. Furthermore, because the master secret key is required to construct f_M from M, the holder of this secret key can recover x. Thus, as with most other functional encryption schemes, this protocol does not provide a solution due to the requirement that the user hold a secret key in order to compute the desired functionality.

However, this quadratic functional encryption scheme has been used to calculate the output of a neural network with one hidden layer (and the activation function $x \mapsto x^2$) [80]. Recall that this scheme assumes that the weights of the neural network are not secret. Furthermore, if the dimensions of the weight matrices network are sufficiently high, anyone in possession of the neural network output and weights of the neural network can invert the calculation and calculate the input.

3.3. Function-Hiding Inner Product Encryption

In this section, we discuss an unsuccessful attempt to adapt current inner product encryption schemes (i.e., those that compute $f_{\boldsymbol{c}}(\boldsymbol{x}) = \boldsymbol{c} \cdot \boldsymbol{x}$) to our use case. Traditionally, it is $\boldsymbol{x}$ that must remain private, and $\boldsymbol{c}$ can be public. It is our desire to produce a scheme that outputs $f_{\boldsymbol{c}}(\boldsymbol{x})$ without revealing $\boldsymbol{c}$.

We first note that these schemes generally assume at a small discrete log is computable. That is, most schemes include a step where, given a group $\mathcal{G}$ and a generator $g \in \mathcal{G}$, it is required to calculate x from g^x. This "discrete log problem" is generally assumed to be difficult, with the brute force approach being among the fastest methods available to recover x. However, when x is restricted to be in a small subset, it is feasible to use the brute force approach to deduce x.

We present our exposition based off the inner-product encryption scheme `IPE` by Abdalla et al. [1], although the concerns we raise transfer to many other inner-product schemes as well. Abdalla et al. construct an `IPE` as follows:

- **Setup.** Given a security parameter λ, $\texttt{IPE.KeyGen}(1^\lambda)$ produces a group $\mathcal{G}$, a prime p, and a generator $g \in \mathcal{G}$. Furthermore, we set the master secret key `IPE.msk` as $\boldsymbol{s} \in \mathbb{Z}_p^\ell$, where

each of the entries of $\boldsymbol{s}$ are chosen randomly from $\mathbb{Z}_p$. We set the master public key $\texttt{IPE.mpk}$ as $(h_i = g^{s_i})_{i\in[\ell]} \in \mathcal{G}^\ell$.

- **Encryption.** Given an input $\boldsymbol{x} \in \mathbb{Z}_p^\ell$, we first draw r randomly from $\mathbb{Z}_p$, and set the encryption of $\boldsymbol{x}$ to be $\boldsymbol{ct} = (\texttt{ct}_0, \{\texttt{ct}_i\}_{i\in[\ell]})$, where $\texttt{ct}_0 = g^r$ and $\texttt{ct}_i = h_i^r \cdot g^{x_i}$.
- **Key derivation.** Given some input $\boldsymbol{c} \in \mathbb{Z}_p^\ell$, define the secret key $\texttt{IPE.sk}_{\boldsymbol{c}}$ as $\texttt{IPE.msk} \cdot \boldsymbol{c}$.
- **Decryption.** Given a ciphertext $\texttt{ct}$ for an input $\boldsymbol{x}$ and a secret key $\texttt{sk}_{\boldsymbol{c}}$ associated to $\boldsymbol{c}$, calculate
$$g^{\boldsymbol{c}\cdot\boldsymbol{x}} = \frac{\prod_{i\in[\ell]} \texttt{ct}_i{}^{c_i}}{\texttt{ct}_0{}^{\texttt{sk}_{\boldsymbol{c}}}}.$$

We note three things from this definition. First, the result of decryption is the value $g^{\boldsymbol{c}\cdot\boldsymbol{x}}$, so it is still necessary to solve a discrete log problem. Second, decryption requires that the value $\boldsymbol{c}$ be available in plaintext. For our use case, this "$\boldsymbol{c}$" would correspond to a function parameter, which we would like to keep secret. Lastly, we simply cannot switch the roles of "$\boldsymbol{c}$" and "$\boldsymbol{x}$" – this would require using the master secret key to encrypt or encode the user input. Knowledge of the master key would then allow the user to recover the secret parameter.

There are other inner product encryption schemes which do not require $\boldsymbol{c}$ or $\boldsymbol{x}$ to be publicly known. However, these schemes all require a secret key to encrypt/encode both $\boldsymbol{c}$ and $\boldsymbol{x}$. These secret keys are usually not the same, but one can derive one from the other. As such, they are not appropriate for our use case.

4. OBFUSCATION PRIMITIVES

In this chapter we discuss various primitives that fall under the general umbrella of "obfuscation". The primitives discussed in this section often do not provide security when used alone; they must be used in conjunction with other techniques to provide security in their desired use case.

4.1. Garbled circuits

Garbled circuits provide perhaps the most notable example of a primitive used to obfuscate computation in cryptographic contexts. Our goal in this section is merely to describe the features and aspects of garbled circuits that are relevant to our use case; see [88, 89] for their construction and analysis for Boolean circuits and [7] for arithmetic circuits. The idea behind garbled circuits is to "garble" each gate in a circuit by replacing each gate by a gate which encodes the gate's functionality but whose inputs and outputs have been suitably randomized and are now given as multi-bit "labels". For example, assume we wish to garble a binary gate for the binary operation op. For each of our two binary inputs x_i (where $i \in \{0,1\}$ and $x_i \in \{0,1\}$), we first randomly assign labels $L_i^{x_i}$ to the inputs x_i, and then encode the gate by storing the values $Enc(L_{\text{out}}^{x_0 \text{ op } x_1}; L_0^{x_0} || L_1^{x_1})$, where $Enc(m;k)$ denotes the result of using a symmetric encryption scheme to encrypt m using key k. Given input labels $L_i^{x_i}$, a user can then decrypt the corresponding entry in the table to obtain the appropriate output label.

The outputs of the final gates of a garbled circuit are often not garbled, allowing a user to obtain the result of evaluating the circuit in the clear. Because the inputs and outputs of all gates in the circuit (with the possible exception of the outputs of the final gates) have been randomized, an adversary cannot deduce the functionality of the circuit by observing only the descriptions of each gate.

In practice, garbled circuits have a number of operational constraints that preclude their use in many applications (although garbled circuits are still used as components in more complex algorithms and in a wide variety of complex applications). For one, a user must have obtained the labels corresponding to their inputs. Furthermore, the security of garbled circuits is severely compromised if the circuit is used more than once (or if the user knows the labels of more than one input).

The standard example given for garbled circuits is that of two millionaires (Alice and Bob) who each wish to know, between the two of them, who is the richest, while simultaneously maintaining a desire to not reveal the actual amount of their individually held wealth. In this scenario, Alice will create a garbled circuit and all relevant labels. Alice will then give Bob the description of the circuit and Alice's labels. Alice and Bob use oblivious transfer [40] as a means for Alice to transfer Bob's input labels without Alice knowing Bob's input and without Bob

deducing the labels of any inputs other than his own. Bob then uses the provided labels and the description of the garbled circuit to calculate the desired output.

There are two challenges that we will encounter when using garbled circuits for our use case. First, the single use nature of these techniques implies that it will be difficult to maintain security given that we wish to perform multiple evaluations of our machine learning analytic. Second, the security of garbled circuits requires that the one evaluating the circuit knows only the labels to one input. In our setting, the evaluator must be able to feed any valid input into the analytic; this implies that the evaluator has knowledge of all available labels.

4.2. Matrix branching programs

A matrix branching program (MBP) is a method used to embed a binary circuit, an arithmetic circuit, or a finite automaton into a series of matrix multiplications. MBPs share much in common with garbled circuits, although the specific means of obfuscation differs between the two techniques. In both techniques, one must first obtain labels or matrices which represent one's input. Furthermore, it is difficult (if not impossible) to derive the output for any input for which one does not have the corresponding input encodings. However, because MBPs are built off arithmetic operations, they are often preferred when one wishes to encrypt the matrix values using homomorphic encryption or graded encoding schemes [66] (cf. Section 3.1).

In particular, any value a may be encoded as the matrix

$$M(a) = \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

There are two key features used by matrix branching programs. First, matrices can be combined through successive multiplication to effectively perform additions and multiplications of certain elements. Second, matrices can be randomized in a way that makes it either difficult or infeasible to extract the non-random components of the information they contain. We now explain each of these features.

A quick calculation will verify that $M(a+b) = M(a)M(b)$. Multiplication is a bit more involved. Let

$$P_1 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad P_2 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad P_3 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$P_4 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad P_5 = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad P_6 = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

One can verify that

$$M(ab) = P_1 \cdot M(a) \cdot P_2 \cdot P_3 \cdot M(b) \cdot P_4 \cdot M(a) \cdot P_5 \cdot M(b) \cdot P_6.$$

Using these primitives, one can encode and compute any non-branching arithmetic circuit.

Next, we explain how to randomize matrices such that it is difficult for an adversary to extract x from $M(x)$. Given a MBP $M_1 \cdot M_2 \cdot \cdots \cdot M_n$, one first constructs $(n+1)$ 3×3 invertible matrices $R_0, ..., R_n$ with entries chosen uniformly at random from $\mathbb{Z}_q$. One then replaces each M_i by $\tilde{M}_i = R_{i-1} M_i R_i^{-1}$. Traditionally, the R_i are used to prevent the MBP from being executed "out of order". However, these matrices can also be used to mask the values hidden in the M_i. Finally, to extract the result of the computation, we must have either $R_0 = R_n = I$, or that there is a decoder which has knowledge of R_0 and R_n.

4.2.1. Weaknesses with the MBP approach

The most significant weakness with the MBP approach is that, With current MPB techniques, one can only calculate arithmetic expressions involving addition, subtraction, and multiplication. As most powerful machine learning decision functions use more complex operations (including transcendental functions, comparisons, and "max"), MBPs are not sufficient.

Furthermore, one can deduce a significant amount of information about the masks used in a MBP. If the MBP is sufficiently small, we can symbolically execute the MBP using software like Mathematica. Otherwise, we can use equations and relationships (discussed below) to recover information about the masks R_i.

If one were to encode the decision function of a machine learning classifier using a matrix branching program, some $\tilde{M}_i$ would represent parameters, which we wish to keep from the user. Other $\tilde{M}_i$ would encode user inputs. This implies that it is necessary for the user to be able to calculate $\tilde{M}_i = R_{i-1} M(x) R_i^{-1}$ for any input x. This will necessarily leak (a surprisingly large amount of) information about R_{i-1} and R_i.

Given random $n \times n$ invertible matrices R and S that are used to mask a user input, the user must be able to calculate $RM(0)S^{-1}$ and $RM(1)S^{-1}$. The matrix $M(1) - M(0)$ contains a single 1 and the second row and first column; all other entries are 0. This allows an adversary to write the second column of R and the first row of S^{-1} in terms of one degree of freedom. The equation $RM(0)S^{-1} = RS^{-1}$ further decreases the number of degrees of freedom. Experimentation with Mathematica suggests that the available equations permit us to reduce the degrees of freedom in R and S down to $(n-1)^2 + 1$.

It is unclear how large of a concern this is. After all, one degree of freedom is enough to mask a value. However, one could envision a scenario with multiple inputs, where a user could "chain together" various equations to further reduce the degrees of freedom. It might be prudent to introduce some arithmetic blinding operations to mitigate this concern. For example, to calculate the equation $\sum_i c_i x_i$, where x_i denote user inputs and c_i denote values to be kept secret, one could construct a MBP for the equation $\sum_i c_i (x_i + r_i) - (\sum_i c_i r_i)$, where the r_i are chosen at random and the sum $\sum_i c_i r_i$ is precomputed.

4.2.2. Potential improvements to MBP encodings

We first discuss two strategies for changing encodings $M(x)$ for user inputs x. Both strategies involve replacing the $M(x)$ discussed above with block diagonal matrices $M_{\mathsf{v}}(x)$. Our first strategy is to introduce blocks of random noise into the encodings of x; these serve primarily to increase the degrees of freedom.

Our second strategy is to use the Chinese remainder theorem to decompose the MBP. We can change the domain of our MBP from matrices over $\mathbb{Z}_q$ for some prime q to matrices over $\mathbb{Z}_N$, where N is the product of two large primes q_1 and q_2. In this setting, our $M_{\mathsf{v}}(x)$ would consist of two blocks. For a (secret) parameter c, we would define $M_{\mathsf{v}}(c)$ to be block diagonal with blocks $M(c+r_1q_1)$ and $M(c+r_2q_2)$, where $r_i \in \mathbb{Z}_N$ is random. The first of these blocks would correctly calculate the MBP modulo q_1, and the second block would correctly calculate the MBP modulo q_2. Bezout's identity could then be used to reconstruct the result of the MBP modulo N (by performing the inverse of the Chinese remainder isomorphism). However one would need to find a method for using Bezout's identity without revealing the coefficients in Bezout's identity, as these could be used to reveal q_1 and q_2.

One might also attempt to use properties of the trace to avoid having to reveal the bookend matrices R_0 and R_n of the MBP. Note that for matrices X and Y, we have $Tr(XY) = Tr(YX)$. Thus, one could find matrices A and B such that $Tr\left(A\left(R_0M(x)R_n^{-1}\right)B\right) = x$. For example, we could choose A and B to first "undo" the randomization from R_0 and R_n^{-1} and then rotate the columns of the resulting matrix $M(x)$ to the right. To hide even more information, the matrix BA can be provided instead of A and B individually.

4.3. Mixed Boolean arithmetic

Mixed Boolean arithmetic (MBA) refers to computations that compute linear combinations of Boolean functions of inputs $\sum_i c_i f_i(\mathbf{x})$, where $c_i \in \mathbb{Z}$, and f_i are functions which operate in a bit-wise fashion on the inputs x_i (e.g., $f_i(x_1,x_2,x_3) = x_2$ or $f_i(x_1,x_2,x_3) = (x_1 \oplus x_2) \wedge (\neg x_1 \vee x_3)$). In this section, we briefly describe how one might use MBA to obfuscate arithmetic expressions; we refer the reader to [41], which we follow closely for our exposition and from which we draw illustrative examples, for a more detailed account.

There are a number of identities in MBA that can be used to obfuscate both the form of an expression and constants used during the computation of an expression. For example, one who wishes to obfuscate the expression $x+15$ (where x is some user input) might first use the identity

$$x+y = 3(x \vee \neg y) + (\neg x \vee y) - 2(\neg y) - 2(\neg(x \oplus y))$$

as a rewrite rule for the expression $x+y$ and subsequently replace all instances of y with the number 15. When multiple such identities are composed, the resulting expression can be rather

difficult to parse. For example, the following expression calculates $y = x \oplus 92$:

$$\begin{aligned}
a &= 229x + 247 \\
b &= 237a + 214 + ((38a + 85) \wedge 254) \\
c &= 3(b + ((-2b + 255) \wedge 254)) + 77 \\
d &= 75((86c + 36) \wedge 70) + 231c + 118 \\
e &= ((58d + 175) \wedge 244) + 99d + 46 \\
g &= 103(2(e \wedge 148) - (e \wedge 255)) + 13 \\
y &= 237(45g + 229(174g \vee 34) + 194 - 247) \wedge 255
\end{aligned}$$

One helpful technique, used above, to hide constants is to replace a constant K with the expression $f^{-1}(E + f(K))$, where f and its compositional inverse f^{-1} are mixed Boolean polynomials, and E is an MBA identity that always equals 0. Perhaps the simplest of such invertible functions are linear (e.g., $f(x) = ax + b$), but one may use more complex functions as well. The work of Klimov and Shamir [63] discusses multiple ways to construct invertible functions and more general sufficient conditions under which functions are invertible. However, [63] does not discuss how to construct the inverse of such a function. In general, we would expect such functions to be difficult, if not impossible to invert (as most polynomials in $\mathbb{R}[x]$ above degree 5 do not have an inverse).

To construct general MBA identities for n variables, one forms a matrix A with 2^n rows whose columns are labeled by the n variables and Boolean functions of the n variables. The columns corresponding to the variables should have all 2^n possibilities for the input variables. MBA identities correspond to elements in the nullspace of A. Thus, in order to find many identities, we'd like the A to have (many) more columns as rows.

For example, the columns of A below correspond to the variables x and y, and the expressions $(x \wedge y)$ and $(x \vee y)$:

$$A = \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right].$$

The element $[1, 1, -1, -1]$ in the nullspace of A corresponds to the identity

$$x + y - (x \wedge y) - (x \vee y) = 0,$$

which allows us to rewrite $x + y$ as $(x \wedge y) + (x \vee y)$.

It should be noted that MBA techniques typically allow one to obfuscate bit-wise operations and addition; indeed, the general procedure outlined above only allows these operations. However, other operations can be likewise obfuscated. For example, [91] notes that

$$xy = (x \wedge y)(x \vee y) + (x \wedge \neg y)(\neg x \wedge y).$$

This identity can be proven by rewriting x as $(x \wedge y) + (x \wedge \neg y)$ (and similarly for y) and using the identity $(\neg x \wedge y) + (x \wedge \neg y) = (x \vee y) - (x \wedge y)$.

There exist two classes of (automated) techniques for reversing MBA obfuscation. In the first of these, symbolic simplification, the attacker attempts to undo the obfuscation by simplifying the obfuscated expression using (a superset of) the rewrite rules used to construct the obfuscated expression. In the second class of techniques, often called bit blasting, the monomial Boolean expressions are first transformed into a standardized canonical form. After this transformation is complete, the way the expression uses the bits of input in order to construct an output can be matched against known functions.

These techniques become less effective as the complexity or number of rewrite rules are increased, or as the number of bits in the input arguments increases. How effective these techniques are is not fully understood or characterized. To give the reader an idea for the runtime of these de-obfuscation techniques, we note that the experiments performed in [41] indicate that it takes about 12 seconds to undo the obfuscation of $x \oplus 92$ using symbolic simplification with all "low-degree" rules provided to the simplification program, and about 61 seconds to undo the obfuscation of $x \oplus 92$ if x is a 12-bit integer using a bit blasting approach.

As might be expected, the simplification algorithms discussed in [41] do not work when not provided with the exact rules used to obfuscate the expression. That is, current simplification techniques do not scale when the pool of rewrite rules used is large [42].

The authors of [41] also recommend using 'constant-specific' rewrite rules, such as $x \oplus 42 = ((x \vee 191) \wedge (x \oplus 106)) + (x \wedge 64)$. We note that this example is more of an example of a higher-order rule (i.e., one with more variables) rather than a constant-specific rule. The rule is actually just an instance of the identity $(x \oplus (y \wedge z)) - ((x \vee y) \wedge (x \oplus z)) - (x \wedge \neg y \wedge z) = 0$ where $y = 191$ and $z = 106$. The use of constants rather than variables does make the simplification more difficult because the expression appears to have 5 terms ($\{x, 42, 191, 106, 64\}$) rather than 3 ($\{x, y, z\}$) and the "correct" relationships between the constants are not easily determined.

4.3.1. Thoughts on multiplication and MBA

MBA is able to perform addition and operations on values performed bit-wise. Multiplication is much less suited to the MBA paradigm. In this section, we discuss various ways to incorporate multiplication into MBA.

We discussed above the derivation of the multiplication rule (derived by rewriting both x and y, and then simplifying):

$$xy = (x \wedge y)(x \vee y) + (x \wedge \neg y)(\neg x \wedge y).$$

We can generalize this technique by rewriting x and y using other MBA identities and simplifying the resulting expression. For example, using the identity

$$x = (x \vee y) + (x \wedge \neg y) - (x \oplus y),$$

one can calculate that

$$xy = 2(x \vee y)^2 + (x \oplus y)^2 + (\neg x \wedge y)(x \wedge \neg y) - 3(x \vee y)(x \oplus y) - (x \oplus y)(x \wedge y) - (x \wedge y)(x \vee y).$$

The strongest rewrite rules for nonlinear functions will use multiple rewrite rules in such a way that terms are canceled. For example, the derivation of the identity

$$xy = (x \wedge y)(x \vee y) + (x \wedge \neg y)(\neg x \wedge y)$$

has terms $+(x \wedge y)^2$ and $-(x \wedge y)^2$. Such cancellation will make it difficult for an adversary to reverse the derivation of the nonlinear rewrite rule. Unfortunately, we do not have a technique that can be used to generate identities that will yield cancellations.

We have a bit more success when we want to compute linear combinations of inputs. For example, assume that it is our goal to find an MBA identity for $ax + by$ for inputs x and y and known constants a and b. We can slightly modify the matrix A (discussed above) used to find corresponding MBA identities. One would multiply the column of A corresponding to x (resp., y) by a (resp., b); the nullspace of A would then correspond to identities with ax and by rather than x and y.

4.3.2. Creating an MBA rewrite rule generator

In this section, we discuss details relevant to creating a program that can generate MBA rewrite rules for expressions. As an example, the program should be able to take the number of desired 'variables' such as $\{x, y\}$, a target expression such as $x \oplus y$, several "input" MBA expressions such as $\{x + y, x \wedge y\}$, and output an MBA identity such as $x + y - 2(x \wedge y) - x \oplus y = 0$ (or several such identities) which contains the target expression.

We desire the ability to create MBA expressions involving several variables. Additionally, the use of high-order MBA expressions (i.e., single MBA expressions involving more than two variables), is also beneficial. Our rewrite rule generator has two main parts. The first part creates a list of MBA expressions from a set of variables. The set of variables, the list of MBA expressions, and the target expression are then provided to the second part of the rewrite rule generator, which creates the truth table corresponding to the expressions and determines the null space of the truth table. From this null space, rewrite rules are created for the target expression.

4.3.2.1. Random MBA expression generator

The MBA expression generator effectively forms a rooted binary tree where each non-leaf node has two child nodes. The non-leaf nodes each represent a binary operation on the node's children. The leaves are random choices of the variables in consideration (and their negation).

For example, the expression generator can produce a 2-level MBA function like $(w \wedge x) \oplus (\neg y \vee z)$ where the XOR is the root, the two XOR branches are AND and OR respectively, and those branches have leaves as w, x, $\neg y$, and z respectively.

For non-leaf nodes, we consider the operations AND, OR, and XOR. Leaf nodes are randomly chosen from the input set of variables (and their negations), with the constraint that no two leaves that share the same parent will involve the same variable. This avoids (to some extent) degenerate expressions like $(\neg y \vee y)$, $(y \vee y)$, $(y \wedge \neg y)$, etc.

4.3.2.2. Rewrite rule generator

The rewrite rule generator first creates the truth table of the input expressions (including the target expression). We then determine the null space of this truth table. When the input expressions are randomly created, it is often the case that the null space of this truth table matrix is trivial or does not contain the target expression. When this occurs, the generator has to be rerun with new inputs until an identity is generated.

Note that an identified basis element of the null space may contain a multiple of the target expression. For example, the generator may produce the identity $x+y-2(x\wedge y)-x\oplus y=0$ given a target expression of $(x\wedge y)$. When working over the ring $\mathbb{Z}/m\mathbb{Z}$, we would require that the modulus m is co-prime to the multiplicity of the target expression so that a multiplicative inverse can be found. For example, when working with 32-bit numbers, we would require that the multiplicity of the target expression be odd.

The issue above is sometimes remedied by using other identities produced for the same input. For example, $-2(x+y)-(\neg x\vee\neg y)+3(x\vee y)+((\neg x)\oplus y)=0$ and
$-3-(x+y)-2(\neg x\vee\neg y)-((\neg x)\oplus y)=0$ can be added to generate a useful identity for $(x\vee y)$ (because all terms in the resulting expression are divisible by 3).

4.3.2.3. Examples

We give some examples of rewrite rules for more complex expressions:

$$\begin{aligned} x+y+z+w \;=\;& 3-((\neg w\vee y)\wedge(\neg w\oplus x))+((w\vee x)\wedge(w\vee\neg y))+((x\vee\neg y)\wedge(x\oplus z))\\ &+((x\vee y)\wedge(y\vee\neg z))-((\neg y\vee z)\wedge(\neg y\oplus z))-((\neg w\oplus x)\wedge(\neg x\oplus z))\\ &+2((\neg w\wedge\neg x)\vee(x\wedge z))+2((w\vee x)\vee(y\wedge z))\\ &+((\neg w\vee\neg x)\oplus(\neg y\vee\neg z))+((x\oplus\neg z)\oplus(\neg w\wedge z))\\ &+((\neg w\oplus\neg x)\oplus(w\oplus\neg y)) \end{aligned}$$

$$\begin{aligned} x+y+z+w \;=\;& -2-4((\neg x\vee\neg z)\wedge(w\oplus\neg z))+((x\wedge\neg y)\wedge(\neg x\oplus z))\\ &-((x\wedge\neg y)\vee(y\oplus\neg z))+4((\neg w\vee x)\vee z)-2((\neg x\wedge\neg z)\oplus(\neg w\vee x))\\ &+2((\neg w\oplus\neg z)\oplus(x\vee z))-2((\neg y\oplus\neg z)\oplus(y\wedge\neg z))\\ &-((x\oplus z)\oplus(w\oplus\neg z)) \end{aligned}$$

We can use such rules for "constant-specific" rewrite rules by replacing y, z, and w above by random constants that sum to c. For example, using 16-bit integers and $c=194$, we obtain:

$$\begin{aligned} w+194 &= 57438+65532(42943\wedge(9903\oplus w))+4(63839\vee\neg w) \\ &\quad +65534(1696\oplus(30799\vee\neg w))+65535(34736\oplus w)+2(57328\oplus\neg w) \end{aligned}$$

Even more generally, we can create a rewrite rule for $w+0$ and then add c to the resulting rule. For example:

$$\begin{aligned} w &= 62326+65532(49086\wedge(44958\oplus w))+4(31719\vee\neg w) \\ &\quad +65534(33816\oplus(27591\vee\neg w))+65535(37944\oplus w)+2(54393\oplus\neg w) \end{aligned}$$

$$\begin{aligned} w+c &= c+62326+65532(49086\wedge(44958\oplus w))+4(31719\vee\neg w) \\ &\quad +65534(33816\oplus(27591\vee\neg w))+65535(37944\oplus w)+2(54393\oplus\neg w) \end{aligned}$$

4.3.2.4. Miscellaneous thoughts

The approach discussed above to finding rewrite rules feels very much like a "guess-and-check" method. That is, it randomly generates functions and subsequently checks if those functions can create an identity for the target expression. If the rewrite rule generator fails, it must be rerun. Another approach that may be worth considering is seeing how "close" of an approximation to the target expression one can get with the given basis functions. One would then continuously iterate on improving the rewrite rule until it is equivalent to the target expression.

Furthermore, for any rewrite rule $w+x+y+z$, we can generate a rewrite rule by $w+x+y$ (or $w+x$) by substituting random constants into "unused" variables and then subtracting the constants from the resulting expression. For example, a rewrite rule for $w+x$ can be generated by taking the rewrite rule for $w+x+y+z$, substituting $y=15$ and $z=35$ into the rewrite rule, and then subtracting 50 (which is the sum of $y=15$ and $z=35$).

We do note, however, that automated tools may be more likely to have the ability to simplify expressions involving fewer variables. That is, there may be a "strength in numbers" principle – expressions with more variables will be harder to simplify. However, it is unclear if an identity with many expressions is better or worse than an identity with fewer expressions. On one hand, including more expressions generates a rewrite rule with more degrees of freedom (and is thus harder to de-obfuscate with traditional automated simplification approaches). On the other hand, generating a very large rewrite rule and substituting constants into many of the variables may provide an attacker with more relationships between the random constants (which may make these constants easier to deduce).

It is also unclear if full randomization of the constants is better than engineering the constants to have a certain Hamming distance. As an example, a situation where it is observed that constants a and b satisfy $a\vee b=a$ and $a\wedge b=b$ reveals that a can be formed from b by flipping some of b's bits from 0 to 1. A "watchdog" step that monitors for situations where such observations may be made may serve to prevent trivial attacks.

4.3.2.5. Rewrite rule generator – version 2

When more variables are added, the time taken to generate a valid rewrite rule increases substantially. This is a result of the fact that it becomes difficult to find a rewrite rule for $w+x+y+z$ (for example) rather than some even multiple of $w+x+y+z$. This section describes an approach that decreases the number of "bad" rules generated.

Our goal is to solve the linear system $\boldsymbol{Ax} = \boldsymbol{y}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is the truth table determined as above and $\boldsymbol{y}$ is the truth table for the target expression (e.g., $w+x+y+z$). In general, $\boldsymbol{A}$ is not a square matrix, so its inverse is not well-defined. One can create a "generalized inverse", $\boldsymbol{A}^g$, such that $\boldsymbol{AA}^g\boldsymbol{y} = \boldsymbol{y}$. As long as $\boldsymbol{A}$ has full row rank, $\boldsymbol{A}^g = \boldsymbol{A}^T(\boldsymbol{AA}^T)^{-1}$ is well-defined. Then $\boldsymbol{x} = \boldsymbol{A}^g\boldsymbol{y} + (\boldsymbol{I} - \boldsymbol{A}^g\boldsymbol{A})\boldsymbol{w}$ for any arbitrary $\boldsymbol{w}$. Currently, we set $\boldsymbol{w} = 0$. One issue is that $\boldsymbol{A}^g\boldsymbol{y}$ may involve divisions that are not well-defined mod 2^n.

Currently, when encountering a system that does not yield a valid rewrite rule, we discard the entire system of equations and start over with new basis expressions. One may be able to increase the success rate by carefully choosing $\boldsymbol{w}$. One may even be able to choose $\boldsymbol{w}$ such that the number of terms on the right hand side of a rewrite rule is reduced or the coefficients are made smaller. Finding an ideal $\boldsymbol{w}$ may be hard to automate efficiently, however.

It should also be noted that the above construction for $\boldsymbol{A}^g$ requires that $\boldsymbol{A}$ have full row rank. Thus, we primarily consider matrices $\boldsymbol{A}$ with more columns (basis expressions) than rows, but our approach seems to work even when that is not the case. If too few basis expressions are provided, the program has a hard time finding usable solutions. This seems to be true more often when the depth of the basis functions is larger. Note that if $\boldsymbol{A}$ does not have full row rank, one can remove linearly-dependent rows from $[\boldsymbol{A}|\boldsymbol{y}]$ to achieve full row rank.

Another optimization (version 2.1 of our rewrite rule generator) is to remove basis functions that are linearly dependent on the other basis functions prior to solving the equation.

4.3.2.6. Performance metrics

Our MBA rewrite rule generator is implemented in Mathematica. We present performance metrics in Tables 4-1, 4-2, and 4-3. We tested the generator on a MacBook with a 2.3 GHz Intel i9 (8 core) processor and 16 GB 2400 MHz DDR4 memory. We denote tests where the program did not finish within ≈ 20 seconds by "DNF". The failure rate is the fraction of attempts that needed to be aborted and tried again with new basis expressions.

In general, all methods and situations either produced a rewrite rule very quickly (in < 1 second) or were unable to generate a rewrite rule in fewer than ≈ 20 seconds. Failure rates were higher when using fewer basis expressions and failure rate is generally correlated with performance. In terms of runtime, it seems that the decreased failure rate when using more basis functions outweighs the potential extra work needed to deal with a larger truth table. Usually, using depth-2 random functions caused solutions to be found faster than when using depth-3 random functions. This trend becomes more pronounced as one uses more variables. Finally, our original rewrite rule generator generally finishes more quickly than versions 2 and 2.1.

Ver	Depth	#Expressions	Runtime (sec)	Avg. #terms in rule	Avg. failure rate
1	3	15	0.04	18.2	0.9
1	2	15	0.01	15.3	0.8
1	3	30	0.02	18.8	0.4
1	2	30	0.009	12.6	0.3
2	3	15	0.06	14.4	0.9
2	2	15	0.04	12	0.8
2	3	30	0.05	27.8	0.4
2	2	30	0.03	28	0.4
2.1	3	15	DNF	N/A	N/A
2.1	2	15	0.3	10.3	>0.95
2.1	3	30	0.03	15	0.5
2.1	2	30	0.04	12.7	0.6

Table 4-1. Performance metrics of MBA rewrite rule generator (with the target expression being a sum of 4 variables)

Ver	Depth	#Expressions	Runtime (sec)	Avg. #terms in rule	Avg. failure rate
1	3	30	DNF	N/A	N/A
1	2	30	0.04	23.7	0.7
1	3	40	0.06	31.8	0.5
1	2	40	0.03	20.8	0.4
1	3	60	0.06	31.8	0.5
1	2	60	0.03	20.2	0.4
2	3	30	DNF	N/A	N/A
2	2	30	0.09	22.4	0.7
2	3	40	0.08	38.1	0.4
2	2	40	0.05	35.4	0.5
2	3	60	0.1	56.6	0.4
2	2	60	0.07	56.3	0.4
2.1	3	30	DNF	N/A	N/A
2.1	2	30	0.5	18.5	0.9
2.1	3	40	0.09	31.9	0.5
2.1	2	40	0.2	21.2	0.7
2.1	3	60	0.1	31.8	0.5
2.1	2	60	0.2	21.5	0.7

Table 4-2. Performance metrics of MBA rewrite rule generator (with the target expression being a sum of 5 variables)

Ver	Depth	#Expressions	Runtime (sec)	Avg. #terms in rule	Avg. failure rate
1	3	60	DNF	N/A	N/A
1	2	60	0.06	34.1	0.5
1	3	64	0.9	63.6	0.9
1	2	64	0.07	28.8	0.5
1	3	80	0.2	63.7	0.5
1	2	80	0.1	28.4	0.5
2	3	60	DNF	N/A	N/A
2	2	60	0.4	45.2	0.5
2	3	64	1	63.7	0.9
2	2	64	0.3	49.4	0.5
2	3	80	0.4	76.7	0.4
2	2	80	0.3	70.7	0.4
2.1	3	60	DNF	N/	N/A
2.1	2	60	0.9	28.9	0.8
2.1	3	64	DNF	N/A	N/A
2.1	2	64	1	31.1	0.8
2.1	3	80	0.3	63.7	0.4
2.1	2	80	0.9	32.5	0.8

Table 4-3. Performance metrics of MBA rewrite rule generator (with the target expression being a sum of 6 variables)

4.3.2.7. Pseudocode

In this section, we give pseudocode for our rewrite rule generators. The term "depth" refers to the depth of the random expressions. For example, $x \oplus z$ has a depth of 1 while $(x \vee \neg y) \wedge (x \oplus z)$ has a depth of 2. The term "vars" refers to the list of independent variables without negation (e.g., $\{x, y, z, w\}$). We assume a working modulus of 2^n. For versions 2 and 2.1, we find it beneficial to use a list of basis expressions that is longer than $2^{\#\text{vars}}$.

We first provide pseudocode to create one random MBA expression. It is understood that these routines will be run independently to create multiple MBA expressions.

To create a depth 1 function:

1. Randomly choose two distinct elements a and b from vars.
2. Negate a with probability 0.5. Assign this to a'.
3. Negate b with probability 0.5. Assign this to b'.
4. Randomly choose a function f from {AND, OR, XOR} and assign this to f.
5. Output $f(a', b')$.

To create a depth $d > 1$ function:

1. Generate two random depth $d-1$ function and assign these to f_1 and f_2.

2. Randomly choose a function from {AND, OR, XOR} and call this f.
3. Output $f(f_1, f_2)$.

Version 1 of the rewrite rule generator, given as input a list L_0 of basis expressions and a target expression E:

1. Append E to L_0 to form the row vector L_2. The last element of L_2 should be the target expression.
2. Create the 2-dimensional truth table for L_2 with matrix representation $\boldsymbol{M}$.
    a) Each row of $\boldsymbol{M}$ corresponds to a different choice of values for the input variables. There will be 2^k rows when using k variables.
    b) Each column of $\boldsymbol{M}$ represents an expression in L_2. There will be m columns, where m is the length of L_2. The last column will represent the target expression.
3. Find a vector $\boldsymbol{x}$ of integers such that $\boldsymbol{M}\boldsymbol{x} = 0 \mod 2^n$ such that the final element of $\boldsymbol{x}$ is exactly 1 (using, for example, Gaussian elimination). If there is no such vector $\boldsymbol{x}$, then the list of basis expressions cannot generate a rewrite rule for the target expression. In this case, abort and start over with different basis expressions.
4. Set $L_2 \cdot \boldsymbol{x} = 0$ and solve for the target expression.

Version 2 of the rewrite rule generator, given as input a list L_0 of basis expressions and a target expression E:

1. Append E to L_0 to form the row vector L_2. The last element of L_2 should be the target expression.
2. Create the 2-dimensional truth table for L_2 with matrix representation $\boldsymbol{M}$.
    a) Each row of $\boldsymbol{M}$ corresponds to a different choice of values for the input variables. There will be 2^k rows when using k variables.
    b) Each column of $\boldsymbol{M}$ represents an expression in L_2. There will be m columns, where m is the length of L_2. The last column will represent the target expression.
3. Remove one linearly dependent row at a time from $\boldsymbol{M}$ until $\boldsymbol{M}$ is full row rank. If $\boldsymbol{M}$ cannot be made full row rank, abort and start over with different basis functions.
4. Assign the last column of (the now full-row-rank) $\boldsymbol{M}$ as column vector $\boldsymbol{y}$. Assign the remaining elements of $\boldsymbol{M}$ as matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not full row rank, abort and start over with new basis expressions.
5. Determine the generalized inverse of $\boldsymbol{A}$: $\boldsymbol{A}^g = \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{A}^T)^{-1}$.
6. Find a vector $\boldsymbol{x} = \boldsymbol{A}^g\boldsymbol{y} + (\boldsymbol{I} - \boldsymbol{A}^g\boldsymbol{A})\boldsymbol{w}$, where $\boldsymbol{w}$ is an arbitrary vector of integers, and all elements of $\boldsymbol{x}$ are well-defined mod 2^n. To be specific, we are trying to find $\boldsymbol{x}$ where none of the elements have even denominators. If no vector $\boldsymbol{x}$ exists, abort and start over with a new list of basis expressions.
7. Set $L_2 \cdot \boldsymbol{x} = 0$ and solve for the target expression.

Version 2.1 of the rewrite rule generator, given as input a list L_0 of basis expressions and a target expression E:

1. Append E to L_0 to form the row vector L_2. The last element of L_2 should be the target expression.
2. Create the 2-dimensional truth table for L_2 with matrix representation $\boldsymbol{M}$.
   a) Each row of $\boldsymbol{M}$ corresponds to a different choice of values for the input variables. There will be 2^k rows when using k variables.
   b) Each column of $\boldsymbol{M}$ represents an expression in L_2. There will be m columns, where m is the length of L_2. The last column will represent the target expression.
3. Let R be the rank of $\boldsymbol{M}$. Without removing the final column, remove one linearly dependent column from $\boldsymbol{M}$ at a time until $\boldsymbol{M}$ has $R+1$ columns.
4. Remove one linearly dependent row at a time from $\boldsymbol{M}$ until $\boldsymbol{M}$ is full row rank. If $\boldsymbol{M}$ cannot be made full row rank, abort and start over with different basis functions.
5. Assign the last column of (the now full-row-rank) $\boldsymbol{M}$ as column vector $\boldsymbol{y}$. Assign the remaining elements of $\boldsymbol{M}$ as matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is not full row rank, abort and start over with new basis expressions.
6. Determine the inverse of $\boldsymbol{A}$: $\boldsymbol{A}^{-1}$.
7. Calculate $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{y}$ such that all elements of $\boldsymbol{x}$ are well-defined mod 2^n. To be specific, we are trying to find $\boldsymbol{x}$ where none of the elements have even denominators. If no vector $\boldsymbol{x}$ exists, abort and start over with a new list of basis expressions.
8. Set $L_2 \cdot \boldsymbol{x} = 0$ and solve for the target expression.

5. MULTI-PARTY COMPUTATION

5.1. Secret sharing

Secret sharing in general seeks to divide data among n shares such that the data can be efficiently reconstructed given k of these n shares. Furthermore, possession of fewer than k of these shares should provide no information about the underlying data [16]. Traditionally, each of these shares is distributed to one of n parties participating in the protocol. Unlike homomorphic encryption, there are no encryption keys in secret sharing schemes. Instead, the shares of data serve as both the encryption and the key.

5.1.1. Shamir secret sharing

In Shamir secret sharing [82], shares are defined as points $\{(i, f(i))\}_{i\neq 0}$ along a polynomial f. f is constructed so that $f(0)$ is the shared secret and the remaining coefficients are drawn uniformly at random from the same domain as the secret. This domain is typically $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ for some prime q. The prime q must be chosen to be greater than the domain required by the magnitude of the secrets and the number of shares. In order to rebuild the secret, one must possess a number of shares greater than the degree of the polynomial f. The most common and intuitive method to deduce $f(0)$ from these shares is Lagrange polynomial interpolation.

This ability to select the degree of the polynomial f independently from the number of players in the scheme permits a threshold scheme for the shares of the secret. That is, should one desire that k of n shares be necessary to rebuild the secret, one chooses f to have degree $k-1$. It is then possible to generate n shares while maintaining the ability to rebuild the secret from any set of at least k of these shares. Furthermore, with at most $k-1$ shares, it is impossible to reconstruct the polynomial or to gain any information about the secret. This is formally referred to as a (k,n) threshold scheme.

Constructing shares. Given a secret s and a target degree $k-1$, one constructs a polynomial f via:

$$f(x) = (c_{k-1}x^{k-1} + c_{k-2}x^{k-2} + \cdots + c_2x^2 + c_1x + s) \mod q,$$

where the c_i are sampled from a uniform random distribution on the domain of the secret. For $i \in \{1,...,n\}$, the share s_i is defined as $s_i = (i, f(i))$.

Rebuilding Secrets. One can use Lagrange polynomial interpolation to reconstruct a secret from shares. The general formula for Lagrange interpolation is:

$$f(x) = \sum_{j=1}^{n} y_j \prod_{k=1:k\neq j}^{n} \frac{x - x_k}{x_j - x_k}.$$

Thus, to recover the secret, we calculate:

$$s = f(0) = \sum_{j=1}^{n} y_j \prod_{k=1:k\neq j}^{n} \frac{-x_k}{x_j - x_k}.$$

Addition. In the Shamir secret Sharing scheme, addition is "free" in that no communication is required between players, and no lengthy local computations are necessary. For two values, a and b, whose associated shares are $\{(i,a_i)\}$ and $\{(i,b_i)\}$, one can calculate shares for $c = a+b$ via $(i,c_i) = (i,a_i+b_i)$. Correctness of this procedure is a result of the fact that if f and g are polynomials of degree $k-1$ with $f(0) = a$ and $g(0) = b$, then $(f+g)$ is a polynomial of degree $k-1$ with $(f+g)(0) = a+b$.

Multiplication. Because Shamir secret sharing is inherently a linear scheme, multiplication is a somewhat complex process. Unlike the addition procedure, we cannot simply multiply values of shares. Assume that we have values a and b and polynomials f and g of degree $k-1$ with $f(0) = a$ and $g(0) = b$. Then the polynomial $(f \times g)$ satisfies $(f \times g)(0) = ab$, but $(f \times g)$ now has degree $2k-2$. Furthermore, the coefficients of $(f \times g)$ are no longer random, and thus leak information.

The multiplication procedure in Shamir secret sharing uses the shares of $(f \times g)$ to construct a polynomial h of degree $k-1$ such that $h(0) = ab$ and such that the non-constant coefficients of h are distributed uniformly at random. We refer the reader to [47] for an exposition of multiplication in Shamir secret sharing. It is important to note that, in order to reduce the degree of $(f \times g)$, the multiplication procedure requires that $n \geq 2k-1$.

5.1.2. Additive Secret Sharing

Additive secret sharing decomposes values into shares such the sum of the shares is the secret value. The underlying security is dependent on the fact that the sum (modulo q) of a fixed value (the secret) and a uniformly randomly selected value is uniformly random. In this context the sum is therefore unconditionally secure since any adversary, unbounded by limits on computational power, can do no better than also simply randomly guess at what the two original values may have been.

Many operations are much more computationally efficient in additive secret sharing than in Shamir secret sharing. However, unlike in Shamir secret sharing, when using additive secret sharing, all shares are required to recover the secret value.

Constructing Shares. Given a secret s and a number of shares n, one first generates $n-1$ values $s_1, \ldots, s_{n-1}$ uniformly at random from $\mathbb{Z}_q$. One then sets $s_n = s - \sum_{i=1}^{n-1} s_i \mod q$. The resulting s_i are the additive shares for the secret s.

Rebuilding Secrets. One can rebuild the secret s from the shares s_i using the identity $s = \sum_{i=1}^{n} s_i \mod q$.

Addition. Shares for a sum $a+b$ can be computed by adding the corresponding shares of a and b. That is, if one has shares $\{s_i\}$ for a and $\{s_i'\}$ for b, then $\{s_i + s_i'\}$ are shares for $a+b$.

Multiplication. The protocol for multiplication for additive secret sharing is again more complex than the protocols for other operations. We provide the basic idea here and refer the reader to [16] for a more complete exposition of the protocol.

We assume shares $\{a_i\}$ for a and $\{b_i\}$ for b (such that $a = \sum_{i=1}^{n} a_i$ and $b = \sum_{j=1}^{n} b_j$). Note that $ab = \sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j$. The multiplication protocol computes additive shares for each of these summands independently among various 2-player subsets of the n players. Note that this forces $n \geq 3$ when performing multiplication using additive secret sharing.

When $i \neq j$, we have players i, j, and k compute shares for $a_i b_j$ via

$$a_i b_j = [a_i(b_j + r_j)] + [-r_j(a_i + r_i)] + [r_i r_j],$$

where r_i and r_j are random elements generated by player k and transmitted to player i and j. The bracketed values in this expression denote the additive shares for players i, j, and k, respectively.

5.2. Machine learning using MPC

MPC techniques have been used to provide private evaluation of machine learning classifiers, including hyperplane classifiers, Naive Bayes classifiers, decision trees and random forests, and support vector machines [23, 65, 79, 85]. Many of the techniques introduced in these works rely on other cryptographic primitives, including homomorphic encryption, oblivious transfer, and secret sharing.

As with other MPC methods, the security of the protocols introduced in these works relies on the inability of an adversary to obtain sensitive information from all parties involved in the protocol. Thus, these techniques are insecure when run on a single computational device but could provide value in a distributed environment. Furthermore, many MPC-based methods for evaluating ML models focus primarily on the privacy of the user's input data rather than the privacy of the ML model. However, due to the nature of the primitives used in many of these MPC-based approaches, there are often privacy guarantees for the model's parameters as well.

5.3. Useful aspects of MPC

Unlike many of the other techniques discussed in this report, MPC generally permits a wider variety of expressions to be computed. For example, the work of Nishide and Ohta [71] allows one to efficiently perform equality and comparison tests (among other operations, including unbounded fan-in or and prefix-or). This work was later improved by Reistad and Toft [77, 76].

Bogdanov et al. [17] introduces additional protocols that are particularly useful for data mining applications, including those to find the most significant non-zero bit position, to perform bit extraction, to perform division by a public constant, and to perform (integer) division on two shared values. It should be noted that the techniques of [17] work only when using additive secret sharing with three shares. How well these techniques can be generalized is a matter of future research.

One of the main dilemmas encountered when attempting to apply cryptographic techniques to secure data analytics is that of number representation. Most techniques discussed in the report apply to numbers represented as standard n-bit integers. However, in the realm of machine learning, floating-point numbers are used almost exclusively to represent numbers. Work of Kamm and Willemson [59] develop computational techniques for performing floating-point operations in the MPC context. In their work, they separately represent the sign bit, significand, and exponent (which can each be represented as integers) and develop protocols for addition, multiplication, bit shifting, inversion, calculating the square root, and exponentiation ($x \mapsto e^x$). This work succeeds (where many others fail) in defining protocols for the more exotic functions (i.e., inversion, calculating the square root, and exponentiation) for two primary reasons. First, the outputs of these functions are rarely integers, so schemes based on standard integers cannot even represent the output of these functions. Second, due to the choice to constrain the significand to lie in the interval $\left[\frac{1}{2}, 1\right)$, computational techniques that use a series to calculate an approximation of a function are more likely to succeed due to the high probability that the interval $\left[\frac{1}{2}, 1\right)$ (and hence, the significand) will lie in the radius of convergence for the Taylor series of the function of interest. This allows floating-point computations that are not possible with fixed-point representations.

5.4. Experiments with SCALE-MAMBA

This section documents experiments with the SCALE-MAMBA framework published by KU Leuven [64]. SCALE (Secure Computation Algorithms from LEuven) is an implementation of many secure multiparty primitives that is bundled together with the high level language MAMBA (Multiparty AlgorithMs Basic Argot), which is designed to interface with the framework. SCALE-MAMBA allows one to write an arbitrary program in a simple style similar to python. This program is then complied into an executable which can be handled by the SCALE framework. The capabilities of the system are much more broad than many such frameworks publicly available. It is built on the SPDZ basic approach to SMC, using oblivious transfers to generate shared correlated randomness in a precomputation phase, then using secret sharing in an online phase to evaluate the functionalities of interest.

Table 5-1. SCALE-MAMBA Performance Tests With Integer Operands

Function	Number of Operations	Total Time (s)	Approx. Time online (s)	Approx. Time per operation (s)
Addition	10	444	60	6.0
Addition	100	474	90	0.9
Addition	1000	460	76	0.1
Multiplication	10	482	98	9.8
Multiplication	100	443	59	0.6
Multiplication	1000	480	96	0.1
Dot Product	10	493	109	10.9
Dot Product	100	493	109	1.1
Dot Product	1000	500	116	0.1
Comparison	10	493	109	10.9
Comparison	100	498	114	1.1
Comparison	1000	fail-r		

The framework supports many data types of interest and includes interesting functions. Data types of interest include finite field elements, fixed point numerical types, and floating point numerical types. Functions included in the framework include standard basic operators such as addition and multiplication, as well as much more complicated functionalities such as division, exponentiation with private exponents, logarithms, and trigonometric functions (for some data types).

In order to explore the utility of the framework, a series of tests were executed testing all these data types with a set of functionalities of interest each for differing numbers of operations nested in a worst-case composition to find the limits of the framework's capabilities. In addition to the built-in functionalities of interest, scripts were written to extend the framework with new functionalities of potential interest. The additional functionalities added are a means to calculate an arbitrary base to the power of a private exponent (the framework implements this only with base 2), the soft sign and sigmoid activation functions, as well as a max function. For each function and data type combination (as far as possible/applicable), the run-time was recorded as the wall time from the invocation of the SCALE executable until the return of the completion of that process. All tests were executed via emulation of multiple parties in multiple sub-processes on one machine, thus network latency is not a significant factor in these results. When tests for particular combinations grew large, failures were recorded. We denote a compile time error (translating the MAMBA code into a SCALE byte-code executable) as fail-c and a run-time error by fail-r. Normally failures were caused by running out of compute resources on the machine used for these tests. After a failure was encountered tests with larger numbers of operands were not performed, these situations are still included and denoted with N/A. Finally, due to the nature of the framework, the vast majority of the time is dedicated to pre-processing and initialization of the dependencies for the online phase of computation. This is in turn dominated by the execution of a number of oblivious transfers determined by the desired security parameter. Executing a minimal program, sharing and revealing one integer secret, incurred a time of 384s.

For the finite field integer data type, four functionalities were tested: addition, multiplication,

private dot-products, and comparison. The results of these tests are included in Table 5-1. Note that there is significant variance in the performance of the pre-computation; since this is the dominating operation for many of the examples, effects of changes in this phase override increasing the number of required online operations in some situations. One example of this can be seen in Table 5-1 in the rows related to performing 100 or 1,000 additions.

For the fixed point data type the widest set of functionalities were tested due to the robust capabilities of this data type. For fixed point numbers, we tested addition, multiplication, dot products, comparison, softsign, sigmoid, arctangent, max, and exponentiation. This data type makes use of approximation for some functionalities though the greatest error was introduced by truncation required since the default precision of the type is only 20 bits. These results are included in Table 5-2.

The final set of tests were for floating point operands. The current state of the framework does not yet have implementation of all the functionalities supported by fixed point operands, so the same set of tests were again executed as was the case for integer field elements: addition, multiplication, dot products, and comparisons. The results for operands of this type and available functions are included in Table 5-3.

Table 5-2. SCALE-MAMBA Performance Tests with Fixed Point Operands

Function	Number of Operations	Total Time (s)	Approx. Time online (s)	Approx. Time per operation (s)
Addition	10	389	5	0.5
Addition	100	397	13	0.1
Addition	1000	445	61	0.1
Multiplication	10	443	59	5.9
Multiplication	100	442	58	0.6
Multiplication	1000	460	76	0.1
Dot Product	10	460	76	7.6
Dot Product	100	455	71	0.7
Dot Product	1000	502	118	0.1
Comparison	10	471	87	8.7
Comparison	100	492	108	1.1
Comparison	1000	fail-c		
Softsign	10	405	21	2.1
Softsign	100	fail-r		
Softsign	1000	N/A		
Sigmoid	10	420	36	3.6
Sigmoid	100	fail-r		
Sigmoid	1000	N/A		
Arctangent	10	443	59	5.9
Arctangent	100	fail-r		
Arctangent	1000	N/A		
Max	10	417	33	3.3
Max	100	448	64	0.6
Max	1000	fail-r		
Exp	10	449	65	6.5
Exp	100	910	526	5.3
Exp	1000	fail-c		

Table 5-3. SCALE-MAMBA Performance Tests with Floating Point Operands

Function	Number of Operations	Total Time (s)	Approx. Time online (s)	Approx. Time per operation (s)
Addition	10	491	107	10.7
Addition	100	509	125	1.2
Addition	1000	fail-c		
Multiplication	10	424	40	4.0
Multiplication	100	415	31	0.3
Multiplication	1000	fail-c		
Dot Product	10	395	11	1.1
Dot Product	100	452	68	0.7
Dot Product	1000	fail-c		
Comparison	10	403	19	1.9
Comparison	100	556	172	1.7
Comparison	1000	fail-c		

REFERENCES

[1] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In *IACR International Workshop on Public Key Cryptography*, pages 733–751. Springer, 2015.

[2] 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):1–35, 2018.

[3] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In *Annual Cryptology Conference*, pages 500–518. Springer, 2013.

[4] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In *Annual International Cryptology Conference*, pages 333–362. Springer, 2016.

[5] Martin Albrecht and Alex Davidson. Are graded encoding schemes broken yet? `https://malb.io/are-graded-encoding-schemes-broken-yet.html`.

[6] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In *Theory of Cryptography Conference*, pages 528–556. Springer, 2015.

[7] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[8] Seiko Arita and Shota Nakasato. Fully homomorphic encryption for point numbers. In *International Conference on Information Security and Cryptology*, pages 253–270. Springer, 2016.

[9] Louis JM Aslett, Pedro M Esperança, and Chris C Holmes. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *arXiv preprint arXiv:1508.06574*, 2015.

[10] Joonsang Baek, Jan Newmarch, Reihaneh Safavi-Naini, and Willy Susilo. A survey of identity-based cryptography. In *Proc. of Australian Unix Users Group Annual Conference*, pages 95–102, 2004.

[11] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[12] Carmen Elisabetta Zaira Baltico, Dario Catalano, Dario Fiore, and Romain Gay. Practical functional encryption for quadratic functions with applications to predicate encryption. In *Annual International Cryptology Conference*, pages 67–98. Springer, 2017.

[13] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*, pages 1–18. Springer, 2001.

[14] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[15] Allison Bishop, Abhishek Jain, and Lucas Kowalczyk. Function-hiding inner product encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 470–491. Springer, 2015.

[16] Dan Bogdanov. How to securely perform computations on secret-shared data. *MasterâĂŹs Thesis*, 2007.

[17] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[18] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit abe and compact garbled circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 533–556. Springer, 2014.

[19] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015.

[20] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In *Annual Cryptology Conference*, pages 461–478. Springer, 2013.

[21] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography Conference*, pages 253–273. Springer, 2011.

[22] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: a new vision for public-key cryptography. *Communications of the ACM*, 55(11):56–64, 2012.

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

[24] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[25] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *CryptographersâĂŹ Track at the RSA Conference*, pages 391–416. Springer, 2020.

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

[27] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[28] Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. *Journal of Cryptology*, 31(1):202–225, 2018.

[29] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[30] Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based fhe as secure as pke. In *Proceedings of the 5th conference on Innovations in theoretical computer science*, pages 1–12, 2014.

[31] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

[32] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. In *International Conference on Fast Software Encryption*, pages 474–493. Springer, 2016.

[33] 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.

[34] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*, pages 3–33. Springer, 2016.

[35] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 377–408. Springer, 2017.

[36] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[37] 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.

[38] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[39] Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay. Functional encryption for inner product with full function privacy. In *Public-Key Cryptography–PKC 2016*, pages 164–195. Springer, 2016.

[40] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[41] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools*. PhD thesis, 2017.

[42] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 27–38, 2016.

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

[44] Shuhong Gao. Efficient fully homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2018:637, 2018.

[45] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–17. Springer, 2013.

[46] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Functional encryption without obfuscation. In *Theory of Cryptography Conference*, pages 480–511. Springer, 2016.

[47] Rosario Gennaro, Michael O Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, 1998.

[48] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[49] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In *Theory of Cryptography Conference*, pages 498–527. Springer, 2015.

[50] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92. Springer, 2013.

[51] 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.

[52] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564, 2013.

[53] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 553–562. IEEE, 2005.

[54] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from lwe. In *Annual Cryptology Conference*, pages 503–523. Springer, 2015.

[55] Thore Graepel, Kristin Lauter, and Michael Naehrig. Ml confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer, 2012.

[56] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[57] Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults. In *ACM Workshop on Digital Rights Management*, pages 16–31. Springer, 2002.

[58] D Kalyani and R Sridevi. Survey on identity based and hierarchical identity based encryption schemes. *International Journal of Computer Applications*, 134(14):0975–8887, 2016.

[59] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[60] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. Shield: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Transactions on Computers*, 65(9):2848–2858, 2015.

[61] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):83, 2018.

[62] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J Wu. Function-hiding inner product encryption is practical. In *International Conference on Security and Cryptography for Networks*, pages 544–562. Springer, 2018.

[63] Alexander Klimov and Adi Shamir. A new class of invertible mappings. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 470–483. Springer, 2002.

[64] KU Leuven. SCALE-MAMBA Software. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`.

[65] Sven Laur, Helger Lipmaa, and Taneli Mielikäinen. Cryptographically private support vector machines. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 618–624, 2006.

[66] Kevin Lewi, Alex J Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching

programs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 981–992, 2016.

[67] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178, 2016.

[68] Fermi Ma and Mark Zhandry. The mmap strikes back: obfuscation and new multilinear maps immune to clt13 zeroizing attacks. In *Theory of Cryptography Conference*, pages 513–543. Springer, 2018.

[69] Paulo Martins, Leonel Sousa, and Artur Mariano. A survey on fully homomorphic encryption: An engineering perspective. *ACM Computing Surveys (CSUR)*, 50(6):1–33, 2017.

[70] Wil Michiels, Paul Gorissen, and Henk DL Hollmann. Cryptanalysis of a generic class of white-box implementations. In *International Workshop on Selected Areas in Cryptography*, pages 414–428. Springer, 2008.

[71] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *International Workshop on Public Key Cryptography*, pages 343–360. Springer, 2007.

[72] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[73] 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.

[74] Zhi Qiao, Shuwen Liang, Spencer Davis, and Hai Jiang. Survey of attribute based encryption. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014.

[75] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[76] Tord Ingolf Reistad. Multiparty comparison-an improved multiparty protocol for comparison of secret-shared values. In *International Conference on Security and Cryptography*, volume 1, pages 325–330. SCITEPRESS, 2009.

[77] Tord Ingolf Reistad and Tomas Toft. Secret sharing comparison by transformation and rotation. In *International Conference on Information Theoretic Security*, pages 169–180. Springer, 2007.

[78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[79] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[80] Théo Ryffel, Edouard Dufour Sans, Romain Gay, Francis Bach, and David Pointcheval. Partially encrypted machine learning using functional encryption. *arXiv preprint arXiv:1905.10214*, 2019.

[81] Amit Sahai and Hakan Seyalioglu. Worry-free encryption: functional encryption with public keys. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 463–472, 2010.

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

[83] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *Theory of Cryptography Conference*, pages 457–473. Springer, 2009.

[84] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[85] David J Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. Privately evaluating decision trees and random forests. *Proceedings on Privacy Enhancing Technologies*, 2016(4):335–355, 2016.

[86] Brecht Wyseur. White-box cryptography, 2011.

[87] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

[88] Sophia Yakoubov. A gentle introduction to yaoâĂŹs garbled circuits, 2019.

[89] 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.

[90] Shushan Zhao, Akshai Aggarwal, Richard Frost, and Xiaole Bai. A survey of applications of identity-based cryptography in mobile ad-hoc networks. *IEEE Communications surveys & tutorials*, 14(2):380–400, 2011.

[91] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for NGNA renewable security. *NCTA-The National Show*, 2006.

DISTRIBUTION

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	D. Chavez, LDRD Office	1911	0359

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Jacek Skryzalin	05646	jskryza@sandia.gov
Kenneth Goss	05952	kgoss@sandia.gov
Benjamin Charles Jackson	05646	bcjacks@sandia.gov
Hamilton E. Link	05962	helink@sandia.gov
William A. Zortman	05646	wzortm@sandia.gov
Marian C. Jackson (“Chrisma”)	05970	mcjacks@sandia.gov
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.