A STANDARD API FOR RLWE-BASED HOMOMORPHIC ENCRYPTION

Michael Brenner, Wei Dai, Shai Halevi, Kyoohyung Han, Amir Jalali, Miran Kim, Kim Laine, Alex Malozemoff, Pascal Paillier, Yuriy Polyakov, Kurt Rohloff, Erkay Savas, Berk Sunar

Motivation

Secure computing techniques such as Ring-LWE-based Homomorphic Encryption[1] (HE) offer the possibility of general computing on data while the data remains encrypted. Practical and usable homomorphic encryption would lead to a sea change in outsourced (cloud) computation on privacy-critical data, and enable a number of application scenarios that are currently either impossible due to technical or legal reasons, or are possible but rely on costly and time-consuming legal processes. Although homomorphic encryption is still at an early point in its life cycle, there has been consistent, substantive, and rapid progress in making it practical from a performance standpoint. homomorphic encryption is increasingly being developed and applied in real-world applications supported by multiple open-source and proprietary libraries.[2]

Applications of homomorphic encryption fundamentally involve distributed applications where ciphertexts, public keys, and additional low-level information needs to be shared between data providers, encrypted computing hosts, and the desired recipients of the results of the computation. Furthermore, homomorphic encryption applications are built around low-level “circuits”, the designing of which is a complex and error-prone task, analogous to programming in assembly language on a machine with a very limited instruction set. In many cases the developer will need to carefully manage and consider low-level aspects of how the data is organized inside ciphertexts. To make homomorphic encryption practical to use by application developers, these challenges need to be overcome. This is the goal of the API standardization effort.

This white paper is being written with the engagement and leadership of the major teams developing open-source homomorphic encryption libraries. Our intent is to motivate, plan, and begin the standardization of an API for homomorphic encryption, and share our thoughts with interested parties in government, industry, and academia for a broader and more official standardization effort.

---

[1] Homomorphic Encryption includes both Somewhat Homomorphic Encryption (SHE) and Fully Homomorphic Encryption (FHE) variants.

[2] Recent open-source RLWE-based homomorphic encryption libraries include HElib, SEAL, NFLlib and PALISADE.

High-Level Outline

As a first step in overcoming the challenges described above, we need to outline a minimal API that absolutely needs to be designed and standardized, and determine and develop an appropriate level of abstraction. This includes describing (1) a *storage model* to identify what needs to be included to both serialize and deserialize keys, ciphertexts, plaintexts, encryption parameters, and scheme/implementation dependent data, and to support homomorphic computations; (2) an *assembly language* -like representation of homomorphic encryption programs, consisting of low-level library calls. These two topics will form the core of the homomorphic encryption standard.

While useful, a standard storage model and a homomorphic encryption assembly language are unlikely to be enough to enable widespread use of homomorphic encryption by application developers due to the difficulties involved in directly interacting with the libraries. Thus, the next step is to create *circuit compilers* that transform the business logic described by application developers into library calls, optionally optimizing the computation in the process. We view this as describing a *programming model* for homomorphic encryption, representing the business logic layer, describing how application developers interact with circuit compilers, and how the compiler interacts with the libraries to perform its tasks. Our initial position is that the programming model will be increasingly important for later application use, but that it is not yet mature enough for the current standardization effort. Thus, we believe that the programming model discussion should happen outside—yet in tandem with—the development of the first version of the homomorphic encryption standard. We have included a preliminary approach to the programming model in Appendix B.

Our intention is to commence standardization based on a single homomorphic encryption scheme and a reference implementation. However, we hold that a standard API should be sufficiently general to support multiple homomorphic encryption schemes, as naturally expected improvements in schemes are made. In particular, the API should be designed to be flexible and compatible enough to support different application scenarios with reasonable performance metrics. As explained above, our thought is that the API is used in a circuit compilation ecosystem, where high-level programs and both dynamically and statically specified parameters are compiled into execution circuits for the libraries. Additionally, some scheme parameters should be able to be dynamically updated at the commencement of execution for applications adaptation.

As such, we present the high-level components here, and expand on these components below:

1. Storage Model, which includes
    - *Cryptographic Context* to represent all data needed for serialization of deserialization of keys, plaintexts, and ciphertexts. This includes a target homomorphic encryption scheme identifier, scheme variant identifier, library

identifier, encryption parameters such as security level, key weights, and distribution parameters.
    - *Payload Representation* of the keys, plaintexts, and ciphertexts; we include this for standardization, because this information is used by homomorphic encryption program compilers during execution.

2. Homomorphic Encryption Assembly Language
    - *Circuit Description* information, which includes information on the circuits being executed; these circuits can be hand-constructed, or output by tools such as a homomorphic encryption circuit compiler. These circuits include low-level calls to library functions, and can be to some extent library-specific for optimizations.

3. Programming Model (outside of the standard)
    - *Business Logic Layer* information, which includes a representation for the application logic.
    - *Circuit Compiler Description*, which includes information on how a circuit compiler converts the business logic into homomorphic encryption assembly language circuits, executable by a homomorphic encryption library, possibly in a runtime environment.

Note that on purpose we do not discuss program compilation and usage because we do not feel there is yet a need to specify compiler operation. Note also that although many use cases include a focus on multiparty data operations and multiparty homomorphic operations are becoming more widely used, we do not yet begin to standardize these concepts because the security and interaction models needed to commence API standardization are still being formed.

Storage Model

In some sense the storage model API is the most mature, in that it follows existing design patterns for encryption API standards. A possible complication is that multiple homomorphic encryption schemes may eventually need to be standardized. However, an initial solution is to focus on the most widely used homomorphic encryption scheme, notionally the *Fan-Vercauteren (FV) scheme* [FV12], although the *Brakerski-Gentry-Vaikuntanathan (BGV) scheme* [BGV12] is also a viable candidate.

Cryptographic Context

The core storage challenge for lattice cryptography is that of storing elements. A common and generic way of storing elements, i.e. (public/private/evaluation) keys, ciphertexts, and optionally plaintexts in homomorphic encryption is to use arrays, which can be either one-dimensional or multidimensional. The elements of an array can be polynomial coefficients, finite field elements, bits, integers modulo a non-prime modulus, or higher level objects.

The Cryptographic Context captures the state of homomorphic encryption/evaluation session, instructs the compiler how to generate a circuit, and further determines the cryptographic library that provides the instruction set. The Cryptographic Context includes the following parts:

- *Scheme ID*
    - Identifies the homomorphic scheme in use, for example, BGV or FV; the Scheme ID allows any party (compiler, application, etc.) to extract correct scheme dependent parameters based on this ID.
    - Variants of a homomorphic scheme should have distinguishable IDs. For example, "BGV standard" and "BGV variant" can have different APIs, storage structures, and parameters. The functionality of a compiler or an application relies on separation at this level.
    - Multiple implementations of the same scheme may share the same scheme ID if and only if they have identical scheme dependent parameters. For example, "BGV standard" can be implemented by two libraries, where only the low-level execution of instruction sets differ. The same behavior of an application and a compiler has to be ensured. This allows developers to choose compatible alternative implementations and libraries.
- *Scheme Independent Parameters*
    - RLWE parameters, that define the form of ciphertext data, and its serialized format.
    - Plaintext Parameters:
        - Plaintext dimensions, that describe the plaintext size as a matrix (two-dimensional array); the size of the array can be restricted by the particular scheme, encryption parameters, and the user's choice of the plaintext modulus (below).
        - Plaintext modulus: an integer that describes the modulus for the entries of the plaintext matrix.
    - Key Payload, that defines the form and serialization format of keys.
- *Scheme Dependent Parameters*
    - Non-standard information specified per particular scheme or implementation;
    - The Scheme ID identifies the format of the Scheme Dependent Parameters.

Payload Representation

If Cryptographic Contexts can be thought of as metadata for the information being stored, the core storage challenge is the representation of the actual payload of keys, plaintexts and ciphertexts. Standardizing payload representation allows encryption scheme parameters and data elements in both encrypted or plaintext form to be serialized, transported and deserialized in a consistent manner that allows unique decoding across platforms and implementations by various vendors. Functions to serialize and to deserialize encryption scheme parameters, keys, and data pieces should be provided in the cryptographic libraries.

Candidate standard information for payload representation includes:

- Cryptographic Context
    - Allows the extraction of Scheme Dependent Parameters according to the Scheme ID;
- Ciphertext
    - Includes ciphertext parameters, an array containing the ciphertext data, the Scheme ID, and optionally a *noise budget*.
- Plaintext
    - Includes plaintext parameters, an array containing the plaintext data, and the Scheme ID.
- Public Key, Private Key, Evaluation Keys
    - Includes an array containing the key data, and the scheme ID.

Homomorphic Encryption Assembly Language

We believe that defining the “right” homomorphic encryption assembly language representation will be an iterative process, and thus in this white paper we only list the major features we believe such an assembly language should contain, but do not define the architecture or the language format itself.

We imagine a system architecture, where the computation model is defined by arithmetic circuits. The operations that this circuit architecture could support are described below in Appendix A. Initially, the circuit architecture will contain fewer instructions, and as homomorphic encryption compilers improve, we can “push” more of the instructions from the underlying homomorphic encryption libraries to the circuit architecture. As an example, we can imagine an early instruction set only containing EvalAdd, EvalMultiply, and EvalSubtract, with instructions like Relinearization and ModReduce handled by the underlying library. As compilers improve, we can add, say, Relinearize to the circuit architecture.

However, even these early compilers will need access to some information about the underlying homomorphic encryption implementation, such as noise growth and performance characteristics. Otherwise, the compilers will have no metrics to optimize against, limiting their effectiveness. Thus, defining some standard API for noise estimation, etc., is an important first step.

Besides instructions, we need to define how the plaintext data itself is represented. As a first approach, we will assume that all plaintext data is encoded as matrices of integers modulo a plaintext modulus, as was already mentioned above. We take inspiration here from MATLAB, where every value, whether it is a scalar, vector, etc., is represented as a matrix. Each variable also encodes some context information, such as whether it is a plaintext or ciphertext. Ciphertexts additionally need to encode their current noise level and size.

Thus, we consider a program in “homomorphic encryption assembly” as containing the following information:

- Input information:
    - Size of each input;
    - Plaintext range / type of each input;
    - Number of inputs / outputs.
- Circuit representation of program, including a listing of the “instruction set” used in the circuit.
- Meta-information on every wire, which includes:
    - Flag of which wires are ciphertext or plaintext;
    - Information for ciphertext wires
        - Noise estimate;
        - Length of ciphertext;
    - Information for plaintext wires
        - Plaintext space.
- Additional Cryptographic Context information as needed to support every gate execution and input/output ciphertext.

The execution can either be realized as a compiler that takes the homomorphic encryption assembly and outputs a target platform -specific program, or as a virtual machine that processes the homomorphic encryption assembly code directly. Such runtime environments may initially perform additional optimizations to the circuit, although we imagine that in the future these can be pushed to a high-level language compiler.

Next Steps in Standardization

The concepts and ideas in this white paper are but a first step on a long path to standardization of homomorphic encryption. As mentioned above, defining the “right” homomorphic encryption system architecture and assembly language will be an iterative process. We are currently a volunteer community working together with system engineers, compiler writers, and application users to balance inherent design trade-offs for an optimal community solution.

As an initial short-term goal, we are continuing our standardization efforts with regular meetings, establishing a website for information sharing, and creating an opt-in mailing list for interested parties. Our goal for the next meeting is to present a prototype language description, which we could then discuss, debate and iterate on as a community. This will be done in close consultation with paired security and application development subgroups. We identify that we will particularly need to:

- Formalize the Cryptographic Context;
- Formalize the Payload Representations;

- Formalize the Homomorphic Encryption Assembly Language and Circuit Representation;
- Notionally select a homomorphic scheme to focus standardization efforts on;
- Describe noise specifications;
- Describe security parameter specifications;
- Formalize the Programming Model (Appendix B).

Appendix A

Assembly Language for Homomorphic Encryption

Instructions Natively Supported by Compiler

- EvalAdd
    - Computes and outputs the sum of two ciphertexts; or
    - Computes and outputs the sum of a plaintext and a ciphertext; or
    - Computes and outputs the sum of a ciphertext and a plaintext.
- EvalSub
    - Computes and outputs the difference of two ciphertexts; or
    - Computes and outputs the difference of a plaintext and a ciphertext; or
    - Computes and outputs the difference of a ciphertext and a plaintext.
- EvalMult
    - Computes and outputs the product of two ciphertexts; or
    - Computes and outputs the product of a plaintext and a ciphertext; or
    - Computes and outputs the product of a ciphertext and a plaintext.
    - EvalMult does not perform relinearization. In the case of FV and BGV schemes the ciphertext size grows by one element after every multiplication. If key switching is required, Relinearize needs to be called explicitly. A simple compiler might choose to automatically follow every call to EvalMult with a call to Relinearize.
- EvalNegate
    - Computes and outputs the negation of a ciphertext.
- EvalPermuteRow
    - Permutes (e.g. rotates) a row of the underlying plaintext matrix. For instance, rotation by 1 of (1,2,3,...,n-1,n) produces (n,1,2,...,n-2,n-1). Internally, this requires key switching.
- EvalPermuteCol
    - Permutes (e.g. rotates) a given column of the underlying plaintext matrix. For instance, rotation by 1 of (1,2,3,...,n-1,n) produces (n,1,2,...,n-2,n-1). Internally, this requires key switching.

Service Operations

Service Operations are additional housekeeping operations that the user may or may not want the compiler to automatically inject into the circuit. In some cases, for performance reasons the user might want to hand-tune the locations of these operations, or the compiler can perform an optimization step to find the optimal locations.

- Relinearize
    - A unary operation that applies the relinearization operation to a given input ciphertext of arbitrary size (in practice the size can be capped to 4 or 5 if needed). Relinearization changes the size down. The number of size steps that the ciphertext can be relinearized by depends on the evaluation key data.
    - Compiler can determine the evaluation key count and store it along with other parameters in the Cryptographic Context.
    - Relinearization gates are injected, and their locations are determined according to optimization flags by the circuit compiler.
    - An expert user might want to manually edit the locations of the relinearization gates, as the optimization problem can be hard and not correctly solved by the compiler.
    - Relinearization would typically be implemented by the KeySwitch operation.
    - Note: Relinearization has several meanings in literature, and the difference to key switching requires clarification. One suggestion is to have relinearization strictly refer to reducing the power of the secret key required in decryption, and key switching refer to a more general operation. In practice, this type of relinearization seems like the only form of key switching that a user might want to control, perhaps with the exception of proxy re-encryption. This is why KeySwitch is instead listed as a compiler operation.
- Bootstrap
    - A unary operation that resets the noise in a given input ciphertext to a specific level determined by the parameters in Cryptographic Context.
    - Note: Bootstrapping is a very expensive operation in most implementations.
    - The output ciphertext may and may not have same secret key as the input ciphertext.
- ModSwitch
    - Change ciphertext coefficient modulus to smaller or larger value.
    - Can be a performance optimization (e.g. in FV scheme) or a necessary operation after (or as a part of) multiplication (e.g. in BGV scheme).
    - In some cases valuable as final operation in circuits to reduce output size as much as possible to improve networking cost.

Compiler Operations

Compiler Operations are operations that are always controlled by the compiler.

- KeySwitch
    - Switches an input ciphertext from using one secret key to another secret key, without changing the underlying plaintext.
    - Arbitrary size input ciphertexts are allowed, and the size of output ciphertext can also be arbitrary (but at least 2). Note: Relinearize only changes the size of the ciphertext.
    - This can be injected automatically by the circuit compiler when the computation involves operations between two input ciphertext encrypted under different secret keys.

Noise Estimation Operations

- NoiseEstimate
    - Used by the circuit compiler to estimate the noise growth in the current circuit and to guide optimization.

Library-Specific Operations

Each library can further provide optimized implementations of certain higher level operations, which a compiler can inject in a circuit optimization phase, or the developer can use either through “intrinsics”, or direct library calls.

- EvalSquare
    - Computes and outputs the square of an input ciphertext.
- EvalCube
    - Computes and outputs the cube of an input ciphertext.
- EvalDotProduct
    - Computes and outputs the dot product of two ciphertext vectors; or
    - Computes and outputs the dot product of a plaintext vector and a ciphertext vector; or
    - Computes and outputs the dot product of a ciphertext vector and plaintext vector.
- EvalLinearTransformation
    - Computes and outputs the product of two matrix ciphertexts such that the output encrypts the matrix product of the underlying input plaintext matrices.
- EvalDFT
    - Computes and output the Discrete Fourier Transform of an input ciphertext.
- ReRandomize
    - Re-randomizes a given ciphertext. Optionally, this can involve noise randomization operations.

Appendix B

Programming Model

This section describes the programming model. We envisage an “LLVM-esque” architecture as the bridge between front-end languages and the back-end homomorphic encryption library; see above figure. Compilers can then be built to target this architecture, and homomorphic encryption libraries can be built to understand and evaluate programs in this architecture. Thus, the focus of standardization should be this Homomorphic Encryption System Architecture, and associated Homomorphic Encryption Assembly Language, rather than the compiler itself. By standardizing the Homomorphic Encryption System Architecture and associated Homomorphic Encryption Assembly Language, different compilers can be built without the need to understand the underlying homomorphic encryption library being used in the backend, and likewise the underlying homomorphic encryption libraries do not need to understand the high-level language.

The most basic way of programming thus is to directly code in the Homomorphic Encryption Assembly Language that our architecture defines. This would likely be the approach taken by homomorphic encryption experts to get the most efficiency out of their implementation. However, non-experts could program in a higher-level language and rely on the compiler to generate the homomorphic encryption assembly code. We view this workflow as similar to uses of standard (e.g., x86) compilers in the 90s. During that time, most developers would program in a high-level language (say, Java or C++), and rely on the compiler to generate x86 code. However, compilers weren’t yet advanced enough to compete against well-written assembly code, and thus expert developers would often develop directly in assembly to attain the best possible performance.

However, compilers quickly evolved to the point where hand-written assembly is needed only in very special circumstances, with existing compilers often producing code *better* than hand-written assembly. We imagine a similar evolution for homomorphic encryption compilers, where initially experts will continue to program directly in the Homomorphic Encryption Assembly Language. As compilers improve, there will be less of a need to program at this low level.

Although our goal for now is to standardize the Homomorphic Encryption Assembly Language and not the Programming Model, or the high-level compiler, it is a useful exercise to walk through what such a compiler would look like. We can imagine a compiler which takes as input an imperative language and compiles it down to Homomorphic Encryption Assembly Language. Besides the input program, we also need to pass to the compiler (1) the number of inputs and outputs of the function; (2) the plaintext data type; (3) scheme parameters (which allows us to supply noise estimation on the outputs); (4) cost estimation for the various operators. Thus, the compiler will need some “behavioral model” of the underlying homomorphic encryption scheme

being used—in particular, the compiler needs to be aware of the noise growth and performance characteristics of the various operations.

Appendix C

Organizers

Kim Laine	kim.laine@microsoft.com
Kurt Rohloff	rohloff@njit.edu

Contributors

Michael Brenner	brenner@rrzn.uni-hannover.de
Wei Dai	wdai@wpi.edu
Shai Halevi	shaih@alum.mit.edu
Kyoohyung Han	satanigh89@gmail.com
Amir Jalali	ajalali2016@fau.edu
Miran Kim	miran5004@gmail.com
Alex Malozemoff	amaloz@galois.com
Pascal Paillier	pascal.paillier@cryptoexperts.com
Yuriy Polyakov	polyakov@njit.edu
Erkay Savas	savas@njit.edu
Berk Sunar	berk@wpi.edu

References

[FV12] Junfeng Fan and Frederik Vercauteren. *Somewhat practical fully homomorphic encryption*. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *(Leveled) Fully homomorphic encryption without bootstrapping*. In Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012, pages 309–325. ACM, 2012.