REVIEW ARTICLES

Unlocking the Potential of Fully Homomorphic Encryption

By Shruthi Gorantala, Rob Springer, Bryant Gipson

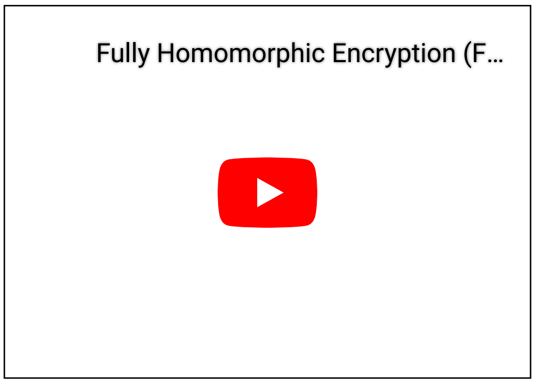
Communications of the ACM, May 2023, Vol. 66 No. 5, Pages 72-81
10.1145/3572832

Comments

VIEW AS: SHARE:

Credit: Andrij Borys Associates, Shutterstock

Modern life is awash with sensitive, private data. From credit cards and banking to conversations, photos, and medical records, our daily experiences are full of software systems that process sensitive data. Despite efforts to build solid data governance strategies, the open nature of the Internet and the difficulty of building robust, secure systems results in regular reports of large-scale data breaches and identity theft. Some applications come with difficult-to-understand data usage agreements, resulting in privacy leaks a user may not be comfortable with. Even with responsible data policy enforcements in place, industry-wide security vulnerabilities are a regular occurrence. Examples include Meltdown[34] and Spectre,[31] hardware vendors who do not properly secure their systems against side-channel attacks[25,32] and more mundane vulnerabilities like SQL injection.[30] Beyond technical issues, social and organizational threats persist in the form of insider risk, social engineering, and software vulnerabilities.[48]



What can be done to address these challenges? An ideal solution would be to encrypt user data and then never decrypt it. While this may sound far-fetched, a Fully Homomorphic Encryption scheme (FHE)[26] does just that. Instead of encrypting data only at rest and in transit, FHE schemes run programs directly on encrypted data, ensuring data stays secure even during computation. In an FHE computation, the program's instructions operate wholly within the encrypted data space, and the final output is only decrypted once it returns to the user's device.

FHE's utility has been limited by poor usability and significant performance costs relative to native execution in the clear. From a usability perspective, FHE programs are specified in terms of primitive instructions that are orders of magnitude slower than the corresponding native operations. Renewed interest across the computing industry has resulted in significant improvements on both fronts. However, there is still much work to do to make it feasible for real-world applications. Microsoft,[a] IBM,[b] Google,[28] and others have all taken steps to improve the usability of FHE, and we will showcase some of our work at Google in this article.

We predict in the next few years, FHE will become viable for many real-world use cases. We can pave the way for FHE to revolutionize data security through investments, hardware-software collaborations, and a strong focus on usability and performance.

Back to Top

Fully Homomorphic Encryption

In an encryption scheme, a message (aka plaintext) is encrypted with a key to produce a ciphertext. The encryption mechanisms guarantee that without the appropriate key, messages cannot be decrypted by malicious parties and information in the message cannot be learned by attackers. Without decryption, a ciphertext is effectively gibberish. Homomorphic encryption is designed to allow computations on the ciphertext while still providing a security guarantee. A homomorphic operation modifies a ciphertext so that the decrypted result is the same as it would have been if the operation were performed on the plaintext. Often called the "Holy Grail" of cryptography, fully homomorphic encryption can perform arbitrary computation on ciphertext.

Most modern FHE schemes are based on a computational problem called "Learning with Errors" (LWE).[39] The LWE problem asks to solve a random system of linear equations whose right hand is perturbed by random noise (See Figure 1). Adding noise hides the message and renders an LWE problem hard to solve. This in-effect implies that the LWE instance becomes hard to distinguish from uniformly random bit strings (see Figure 2). LWE also has reductions from certain lattice problems believed to be hard even for quantum computers to solve.[38] In these LWE based schemes, messages are encrypted, taking advantage of the noise to ensure security guarantees. As long as the noise is sufficiently small, the ciphertext can be decrypted to the correct message.

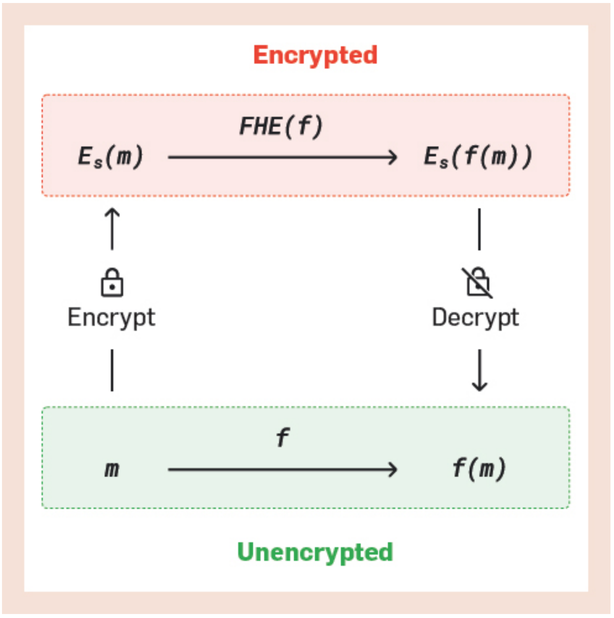
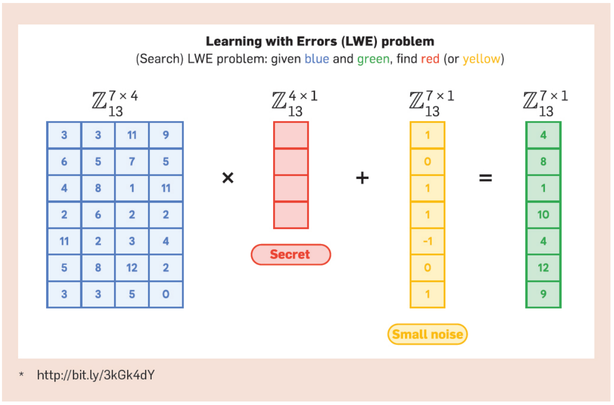


Figure 1. What is homomorphic encryption?



Learning with Errors (LWE) problem
(Search) LWE problem: given blue and green, find red (or yellow)
$\mathbb{Z}_{13}^{7\times 4}$
$\mathbb{Z}_{13}^{4\times 1}$
$\mathbb{Z}_{13}^{7\times 1}$
$\mathbb{Z}_{13}^{7\times 1}$
Secret
Small noise
* http://bit.ly/3kGk4dY

$$\begin{pmatrix} 3 & 3 & 11 & 9 \\ 6 & 5 & 7 & 5 \\ 4 & 8 & 1 & 11 \\ 2 & 6 & 2 & 2 \\ 11 & 2 & 3 & 4 \\ 5 & 8 & 12 & 2 \\ 3 & 3 & 5 & 0 \end{pmatrix} \times \begin{pmatrix} \\ \\ \\ \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ -1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 8 \\ 1 \\ 10 \\ 4 \\ 12 \\ 9 \end{pmatrix}$$

Figure 2. Learning with errors problem.*

With some work, operations such as addition and multiplication can be defined on LWE ciphertexts. However, applying these operations increases the noise. While noise growth is negligible for additions, for multiplication it is not. As a result, only a fixed number of consecutive multiplications can be performed before the noise corrupts the message and decryption becomes impossible. A key innovation in FHE is to reduce the noise by using a technique called "bootstrapping." Bootstrapping homomorphically decrypts a message. In effect, it resets the noise of a ciphertext to a fixed, lower level, allowing further computations to be performed. Bootstrapping is an understandably expensive operation, and much research around FHE today attempts to avoid or accelerate it.

FHE schemes. The concept of homomorphic encryption was first proposed in the 1970s.[41] We've been halfway there for decades with partially homomorphic encryption schemes which allow either addition or multiplication (but not both) to be performed on cipher-text. The Paillier cryptosystem[37] has homomorphic addition but not multiplication, and RSA[40] has homomorphic multiplication but not addition.

The first FHE scheme built on ideal lattices was proposed by Craig Gentry in 2009.[26] The first generation of FHE schemes were slow, requiring at least 30 minutes[46] for a single multiplication. These first-generation schemes convert a somewhat homomorphic encryption scheme (SWHE) into an FHE scheme through bootstrapping. A somewhat homomorphic encryption allows additions and some limited number of multiplications before noise corrupts the ciphertext.

Second-generation schemes (BGV[6] and FV[23]) built on LWE[7] and RLWE[8] focused primarily on improving the efficiency of homomorphic computations using leveled schemes. A leveled homomorphic encryption scheme can evaluate circuits of known (relatively large) depths even without bootstrapping. These schemes introduce the concept of relinearization[6] and modulus-switching[6] as optimizations. Bootstrapping is still available for leveled schemes to support circuits of unlimited depth but not widely used in practice. These schemes also introduce an optimization technique called Single Instruction/Multiple Data(SIMD)-style batching,[43] which reduces ciphertext size blowup by packing vectors of integers into each ciphertext and increases scope for parallel homomorphic computations.

A third generation of schemes (GSW,[27] FHEW,[22] CGGI[14]) focused on improving bootstrapping by several orders of magnitude bringing down the bootstrap time to half a second in FHEW and less than 0.1 seconds in TFHE/CGGI.[16] See Micciancio and Polyakov[35] for a comparison of bootstrapping across these schemes. FHEW[22] also introduced the concept of programmable bootstrapping which allows evaluation of a univariate function during bootstrap. Chillotto et al.[18] extends the idea of programmable bootstrapping to larger plaintext sizes.

A new generation of FHE schemes was introduced in 2017 which constructs a leveled homomorphic encryption scheme to Approximate Arithmetic Numbers (HEAAN) also named as CKKS.[11] CKKS further improves efficiency of BGV/BFV by allowing faster numerical computation by approximation and is more applicable to a wide range of arithmetic applications.

FHE's unique computational challenges. Integrating FHE into an application poses unique challenges from the data model down to the hardware. For example, a program in FHE must be expressed as a circuit. Beyond being an unnatural computation model for average programmers, one consequence is that a homomorphic program must be rewritten to evaluate all paths through the program (in effect losing all branch and bound optimizations). So, one cannot employ conditional jumps or early loop exits. All conditional statements need to be rewritten in the form of a MUX gate. Execution must be truly data independent.

```
if (condition}) { return
return a; condition*a +
} else { (1-condition) *b;
return b;
}
```

In addition to this data-independent computational changes, the choice of an FHE scheme also directly impacts the performance and accuracy of homomorphic programs.

Some schemes are lossy (CKKS[11]), some are suitable for evaluating boolean circuits (TFHE[16]), while some are optimized for numerically heavy computations (BGV[6]). Switching between schemes on the fly is possible, but expensive (CHIMERA[5]). After choosing the scheme, a specific parameter set and encoding scheme need to be chosen to meet a desired security level while minimizing ciphertext size and noise growth. Automatic parameter selection and standardization of API are still open questions and various standardization efforts are still work in progress.

We predict in the next few years, FHE will become viable for many real-world use cases. We can pave the way for FHE to revolutionize data security through investments, hardware-software collaborations, and a strong focus on usability and performance.

FHE computations on encrypted data are significantly slower than those on unencrypted data.[46] Some form of hardware acceleration will be required to compensate. Multiple FHE accelerators are being actively developed, such as Intel's HEXL[3] and Intel HERACLES,[9] MIT/SRI International's F1 accelerator,[24] and CraterLake,[42] and an optical accelerator by Optalysys.[36] While hardware acceleration will certainly deliver huge gains, the need for special hardware adds yet another obstacle to applying FHE at scale.

FHE libraries and compilers. To start implementing an FHE application, a developer can make use of FHE libraries, domain specific compilers or general-purpose compilers. FHE libraries implement the various FHE schemes described above. CONCRETE,[17] TFHE[15] libraries are based on CGGI scheme and FHEW[22] is based on FHEW scheme. CONCRETE, TFHE and FHEW use Ring-GSW internally for bootstrapping. HEAAN[13] implements CKKS scheme and SEAL[c] implements BGV, BFV and CKKS schemes. PALISADE[d] supports multiple schemes including BGV, BFV, CKKS, FHEW, TFHE. Lattigo[e] supports BFV and CKKS. HELib[29] supports BGV and CKKS. See Cheon et al.[12] and Viand et al.[46] for more details of FHE schemes and libraries.

Domain specific compilers focus on a subset of computations such as arithmetic or machine learning. For example, ALCHEMY,[f] Marble[47] and RAMPARTS[2] are all FHE compilation tools based on the BGV or FV schemes, which are good for homomorphic arithmetic operations but suffer from inefficient bootstrapping operations. There is also a growing literature on building FHE compiler tools for specific workloads. For example, nGraph-HE,[4] SEALion,[45] CHET,[21] and EVA[20] all intend to produce efficient and FHE-friendly code for certain machine learning workloads.

General-purpose compilers allow for any computation but are typically slower. The Cingulata toolchain[10] is based on the TFHE scheme and converts C++ into Boolean circuits, performs gate operations, and converts the Boolean operations to FHE binary operations. Encrypt-Everything-Everywhere (E3)[13] also based on TFHE, enables FHE operations, but requires the program to be written with overloaded methods provided by the E3 library. It also requires the user to specify a configuration on the data types used. Most of the compilers above have remained dormant years with no active development.

Multiple benchmarking efforts[g] and standardization efforts[h] are in progress.

The lack of a uniform API and concrete benchmarks makes it a struggle to systematically compare the dizzying array of trade-offs across FHE schemes, libraries, compilers, optimizers, hardware, among others.

Due to these challenges, controlling the heavy cryptographic complexity is a critical requirement for widespread adoption of FHE. Traditional encryption algorithms such as AES and RSA provide security guarantees at rest and in transit. One can abstract away the cryptographic complexity of these schemes by treating them as a serialization and deserialization layer that can be bolted onto any application with effective key negotiation and management. This layer is completely independent of the data types and hence can treat all messages as an array of bits or bits of fixed sized bitwidth bitwords. FHE, on the other hand, provides security guarantees at computation time and inherently depends on the data types as well as computations performed by the program.

Back to Top

The FHE Stack

To streamline FHE engineering, we need better interfaces and abstractions between the layers of an FHE-based application. The main three layers are the application logic—the input programs that need to be converted to use FHE—the cryptosystem implementations, and the hardware.

Having clean interfaces between the three layers both limits development complexity and provides a separation of responsibilities. By limiting complexity, developers can focus on writing application code. Engineers and project managers can also use these abstractions to make principled trade-offs between privacy, latency, and hardware budgets that suit their application's needs. With a good separation of responsibilities, domain experts can make independent progress in each layer to benefit the entire system. For example, cryptographers can improve FHE cryptosystems, hardware engineers on platform-specific optimizations, and compiler experts on improving static analysis tools for optimizing FHE circuits for latency or throughput.

In this view, the LLVM project[i] provides an architectural lodestar for FHE infrastructure. With LLVM, an engineer or researcher can easily experiment with new breakthroughs in different stages of compilation, such as static analysis, register allocation, or hardware optimizations. The layers of abstraction provided by LLVM allow for both the conceptual isolation of those changes and consistent benchmarking across a variety of test applications and target platforms. An improvement to any of these components benefits dozens of compilers built on top of LLVM.

While FHE has novel restrictions absent from traditional compilers inherently tied to its data-independent computational requirements, these same restrictions provide benefits inaccessible to traditional compilers. For example, because an FHE circuit is data independent, its runtime (for a particular hardware target) is static and can be directly optimized at compile time. This avoids an entire class of problems in traditional compilers related to "hot spots" that typically require runtime profiling to understand.[j]

Moreover, because FHE is a nascent technology, it's not clear which improvements will bear fruit in the long term. Establishing a modular architecture early on allows the FHE community to experiment with a variety of cryptosystem backends without rebuilding the other components of the FHE stack from scratch.

We call this idealized vision the FHE stack (See Figure 3). Here, we describe the components of the FHE stack in more detail, loosely broken into the intermediate representation, frontend, middle-end, backend, and hardware adapters.

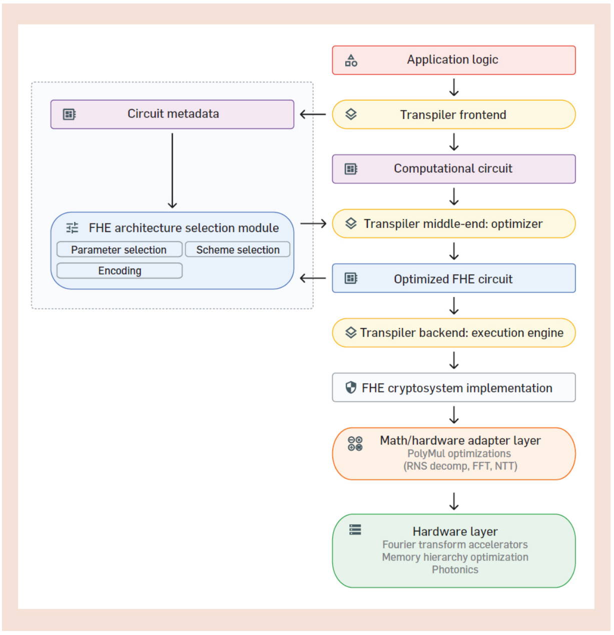

Application logic
Transpiler frontend
Circuit metadata
Computational circuit
FHE architecture selection module
Parameter selection
Scheme selection
Encoding
Transpiler middle-end: optimizer
Optimized FHE circuit
Transpiler backend: execution engine
FHE cryptosystem implementation
Math/hardware adapter layer
PolyMul optimizations
(RNS decomp, FFT, NTT)
Hardware layer
Fourier transform accelerators
Memory hierarchy optimization
Photonics

Figure 3. The FHE stack.

Intermediate representation. The interface between the layers of the FHE stack will be intermediate representations (IRs) of circuits at varying levels of abstraction. These intermediate representations take the form of circuits whose gates represent operations with varying levels of abstraction.

Between the first two layers—the frontend and circuit optimizer—the IR represents a high-level circuit. As with the FHE transpiler we describe later, this IR may represent arithmetic on 32-bit integers as first-class gates. At some point during the middle-end phase, that IR must be converted to a lower-level IR in a manner that depends on the chosen cryptosystem backend. Again, using the FHE transpiler as an example, it might convert 32-bit adder gates into binary ripple-carry adder subcircuits or add them natively with an FHE scheme that supports arithmetic.

It is critical the second IR treats cryptosystem primitives and metadata as first-class entities. This includes gate-level metadata like noise growth that occurs from processing gates of a particular type. It also includes a variety of ciphertext maintenance operations, like bootstrapping and re-linearization, which are periodically required to limit noise growth over the life of the program.

This mirrors the insight of LLVM's IR that requires some carefully chosen high-level metadata to be persisted in the IR directly for the different tools in the FHE stack to perform their jobs well.

Transpiler frontend. The transpiler frontend converts an input program from a source language to a circuit in a high-level circuit IR, optionally emitting additional metadata to define the interface to that circuit in terms of higher-level data types. This component of the stack is largely independent of FHE. However, we include it to highlight that separating the frontend from the rest of the stack—and requiring an IR as the only legal interface between the frontend and the middle-end—forces a clean separation and allows new source languages to onboard to the FHE stack simply by implementing a new frontend.

Establishing a modular architecture early on allows the FHE community to experiment with a variety of cryptosystem backends without rebuilding the other components of the FHE stack from scratch.

FHE architecture selection module. At some point soon after the frontend completes, an abstract FHE scheme and its parameters must be chosen. For example, one could choose to transpile the program into a TFHE cryptosystem backend or a CKKS backend. Within a given scheme, there are a variety of parameter choices that affect runtime, memory requirements, and security. At a high level, this choice is spiritually like a general choice of platform architecture, that is, selecting GPU vs CPU or selecting x86 vs ARM. While there is much variation between different GPUs, most share enough architectural principles that compilers can include general optimizations that are suitable for most GPUs. For this reason, we call this step "FHE architecture selection."

More concretely, this component must decide whether the program will be represented with arithmetic circuits or boolean circuits—only some FHE schemes support each choice. Depending on the choice, different types of optimizations are eligible like SIMD-style ciphertext packing or circuit optimization to minimize the need for bootstrapping.

This choice is abstract, in the sense it is independent of the choice of hardware and the backend library implementing the chosen cryptosystem. For example, two different libraries may implement the CKKS scheme, but at this step the transpiler would only select CKKS in abstract. The optimizations that follow may be specific to CKKS, but not the cryptosystem library implementation details (that will be the role of the backend). As mentioned earlier, choosing a scheme and security parameters is a challenging task that is currently being standardized by the community.

Initial versions of the FHE stack will require human input for this stage. In the long term, this can be automated from high-level metadata like whether the program is latency-sensitive or optimizing for throughput, or whether a polynomial approximation is acceptable.

Transpiler middle-end. The transpiler middle-end converts the high-level circuit IR into an FHE circuit based on the preceding architecture selection step. This resulting lower-level IR has a more direct translation into a particular backend implementation. In the FHE transpiler described later, this step involves booleanizing the high-level IR to convert each boolean gate operation to invocations of the corresponding homomorphic gate operations in the TFHE backend library.

The initial conversion is followed by a series of optimizing passes. In a traditional compiler, the optimizer's goal is to make program runtime fast in a platform-independent manner. This goal is subject to a hard constraint on program correctness, and a secondary objective to keep compilation time decently fast. Like a traditional compiler, the circuit optimization step in the FHE middle-end can be expressed as a series of independent "passes," each matching a particular type of optimization to apply. Not all optimizations apply to every invocation of the transpiler, and they can be enabled or disabled by the client.

However, the FHE circuit optimizer will have distinct differences. A new constraint on top of program correctness is the correctness of the resulting decryption. In particular, the noise accumulated in the LWE ciphertexts by homomorphic operations must not corrupt the underlying message. This depends on when bootstrap operations are applied, or—in the case of leveled HE systems—the depth of the circuit. To allow an optimizer to analyze the circuit and perform the necessary transformations to maintain decryption safety, the operations in the circuit IR should include metadata about its impact on noise growth. This in turn depends on the chosen FHE parameters and the target cryptosystem.

The objective of the optimizer must also make a different trade-off: the performance of the output program will take higher priority due to the afore-mentioned overhead of using FHE, and this may come at the cost of longer compile time or additional input metadata. Moreover, some optimizations may refine the cryptosystem security parameters to improve latency or throughput while maintaining the same security level.

Transpiler backend. The transpiler backend is a series of optimizers and code generators that targets a specific cryptosystem implementation library and hardware backend.

Analogous to register allocation or allocation pooling in traditional compilers, an FHE backend may modify the circuit to take advantage of parallelism specific to the target cryptosystem library and hardware platform. For example, the FHE transpiler includes a scheduler backend that leverages the natural parallelism in the optimized circuit to speed up evaluation.

The algorithms used by backend optimizations can be shared among multiple backends but will not be generally useful to all FHE circuits.

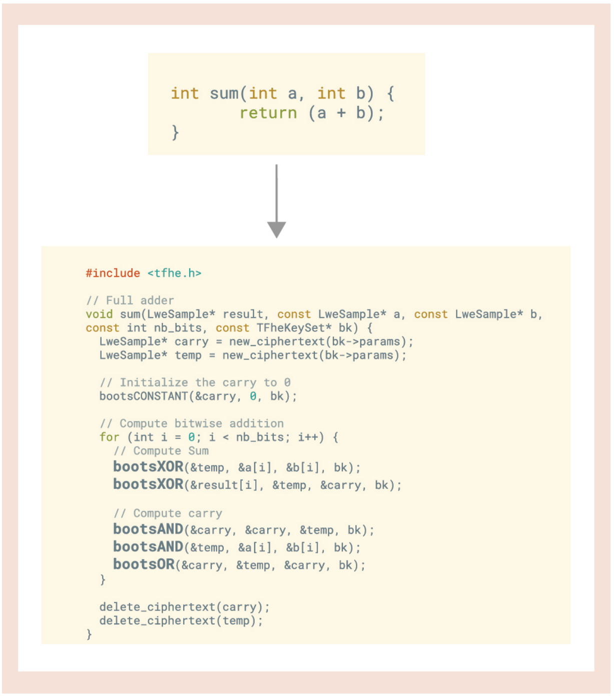
Math/hardware adapter layer. The math/hardware adapter layer is responsible for specializing the implementation of cryptographic primitives to specialized hardware (such as GPU, TPU, ASIC, Photonics). These would come in the form of libraries for operations like fast Fourier transforms (FFT) or number-theoretic transforms (NTT) that implement mathematical operations like modular polynomial multiplication or matrix multiplication common to many cryptosystem backends.

These libraries would be required to be included directly in the crypto-system implementation libraries and would only support specific classes of hardware targets. We primarily highlight the need for common implementations of these primitives that can be shared across cryptosystem backends. Many already exist that are not specific to cryptography, like FFTW[k] for Fourier transforms and NTL[l] for polynomial ring arithmetic.

Back to Top

FHE Transpiler

Recently, Google has been focusing on the abstraction layer between application code and an FHE cryptosystem. We implemented an FHE transpiler[m] that compiles C++ source code that operates on plaintexts into C++ code that operates on ciphertexts. In other words, instead of manually specifying FHE operations as combinations of logic gates, one can instead write code in a subset of C++ (see the section "Limitations" for details), and the transpiler will convert that into an equivalent FHE-friendly C++ function (see Figure 4).



```
int sum(int a, int b) {
        return (a + b);
}
```

```
#include <tfhe.h>

// Full adder
void sum(LweSample* result, const LweSample* a, const LweSample* b,
const int nb_bits, const TFheKeySet* bk) {
  LweSample* carry = new_ciphertext(bk->params);
  LweSample* temp = new_ciphertext(bk->params);

  // Initialize the carry to 0
  bootsCONSTANT(&carry, 0, bk);

  // Compute bitwise addition
  for (int i = 0; i < nb_bits; i++) {
    // Compute Sum
    bootsXOR(&temp, &a[i], &b[i], bk);
    bootsXOR(&result[i], &temp, &carry, bk);

    // Compute carry
    bootsAND(&carry, &carry, &temp, bk);
    bootsAND(&temp, &a[i], &b[i], bk);
    bootsOR(&carry, &temp, &carry, bk);
  }

  delete_ciphertext(carry);
  delete_ciphertext(temp);
}
```

Figure 4. Example transpiler transformation.*

This is a general-purpose compiler and thus employs gate operations on ciphertexts, so we use the TFHE cryptosystem:[16] it exposes a boolean-gate-level API and performs bootstrap operation after every gate operation allowing unlimited computations without noise management. We refer the reader to Gorantala et al.[28] for details on the FHE transpiler. Here, we summarize the parts of the FHE stack addressed by the transpiler.

Design. Inspired by the idealized version of the FHE stack, the FHE transpiler addresses four parts of the FHE stack. Specifically, we chose an IR, we built a frontend for C++, we build a middle-end with optimization passes related to gate parallelism, and a backend for the TFHE cryptosystem.

For an IR we use XLS IR.[n] XLS[o] is a software development kit for hardware design. As part of its toolchain, it provides functionality to compile high-level hardware designs down to lower levels, and eventually to Verilog.[44] This compilation step introduces a flexible intermediate representation (XLS IR) in the form of a computational circuit (see Figure 5).

XLS IR is designed to describe and manipulate low-level operations (such as AND, OR, NOT, and other simple logical operations) of varying bit widths. This interoperates well with FHE programming model as it allows unpacking of datatypes and operations from *n*-bits to series of *k*-bits. The corresponding data flow graphs in XLS allow us to easily represent unpacking, computation, and repacking high-level programming constructs such as arrays, structs, and classes.

The frontend uses XLScc[p] to generate XLS-IR from a C++ input program. The middle-end converts an XLS-IR to a boolean-gate equivalent required by TFHE and makes use of off-the-shelf circuit optimizers such as Berkeley ABC to reduce unnecessary gates. The backend converts the optimized circuit into C++ program that evaluates each gate using the gate bootstrapping API from the TFHE cryptosystem and handles parallelism optimizations along with multiple user-specified execution profiles (single-threaded C++ code or a parallel circuit interpreter). This is paired with a generated C++ interface, exposing a clean API operating on encrypted data.

So far, we have assumed the FHE Architecture Selection module is completely driven by human input, and we hardcode the architecture as the TFHE scheme, binary encoding and default security parameter set provided by TFHE.

FHE transpiler highlights and limitations. The primary highlight of the FHE Transpiler project is its modular architecture and well-defined interfaces which have enabled us to incorporate multiple IRs, middle-ends, backends and cryptosystem implementations. For example, the FHE transpiler now supports the Yosys hardware synthesis tool that compiles RTL to a Verilog netlist (a new IR for FHE transpiler). As noted, the transpiler backend has two execution engines: a single threaded C++ transpiler and the multi-threaded interpreter. Multiple FHE cryptosystems are also supported such as TFHE[14] and PalisadeBinFHE.

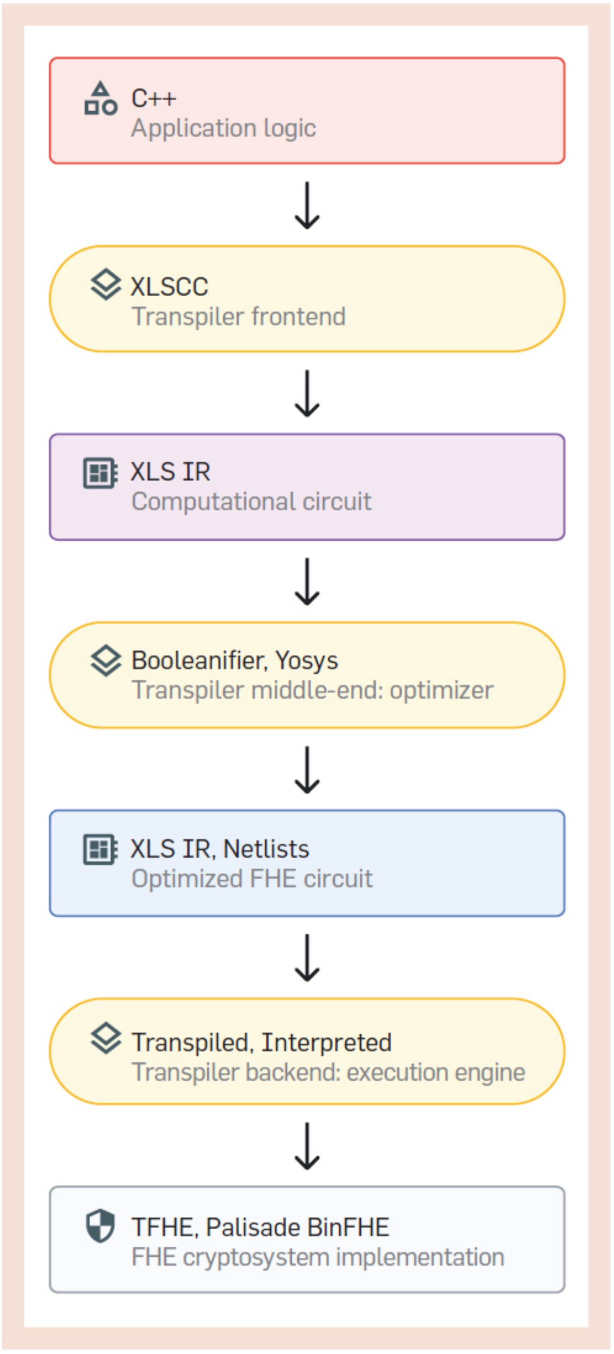


Figure 5. The FHE transpiler.

Our transpiler also automatically generates wrappers with the same API for user-provided data types (for example, classes and structs). This allows for ease of use for encryption/decryption of data on the client-side. Generation of wrapper API facilitates faster client-side development by providing encryption and decryption utilities.

To our knowledge, seamless modular architecture support, fully expressive intermediate representation, generation of wrapper code, and extensibility to multiple cryptosystems have not been implemented in prior works.[10,13] While E3[13] supports multiple FHE cryptosystems, it requires the user to explicitly define a configuration file with data types, sizes and modify significant part of code to use newer data types which FHE transpiler only requires code modification on the client-side for encryption/decryption using the wrapper API. The FHE transpiler also provides the debugging utilities to execute the FHE circuit on plaintext bits.

Due to the nature of FHE and the data-independent programming model, the transpiler does not support dynamic loops, early returns, dynamic arrays, among others. These restrictions can be summarized as a category of programs that require data types to be static and data size to be upper bounded at compile time. The FHE transpiler as it is today carries all the restrictions of the FHE programming paradigm in addition to those imposed by the HLS tools used. Most notable of those is that pointers are not supported. Due to this and similar restrictions, the transpiler does not support multiple features in C++ language including most of the standard library, virtual functions in inheritance, and lambdas.

Roadmap. The IR for an FHE circuit must be enhanced to include ciphertext maintenance operations (as mentioned previously) and the costs they incur. It remains to be explored whether it requires a definition of a new IR or enhancement of existing IRs would suffice. While XLS IR was chosen as the initial candidate for FHE transpiler, MLIR[33] can be a good candidate as it allows for specification of custom operations.

FHE is slowly approaching practicality, but still needs significant engineering investment to achieve its full potential.

Today the FHE transpiler only supports one frontend, C++, and one backend FHE scheme, TFHE. But its modular design naturally allows for additional frontend languages and hardware platforms, such as GPU via the cuFHE[q] library. Additional target cryptosystems in various languages can be supported via the transpiler execution engine. We expect to add additional frontends and backends in the coming years.

The FHE transpiler does not yet address the FHE Architecture Selection Module that assumes a fully composable boolean scheme with gate bootstrapping (such as FHEW or TFHE), boolean encoding, and a predefined set of security parameters are used. Boolean circuits for arithmetic computation incur heavy latencies: adding support for arithmetic schemes and SIMD batching is a natural next step. Addition of optimizers from various domain specific FHE compilers to FHE transpiler project allows for uniform benchmarking (see HEBench) across schemes, libraries, optimizers, hardware, and so on. Efforts toward standardization of API across FHE libraries allows for easier integration into FHE transpiler and faster research and benchmarking.

Back to Top

FHE for Everyone

As Pascal Pallier highlighted in a talk to FHE.org,[r] fully homomorphic encryption is today where deep learning was 10 years ago. FHE is slowly approaching practicality, but still needs significant engineering investment to achieve its full potential. When deep learning emerged, only a small group of researchers could implement useful systems, driven largely by folk knowledge around optimizers, regularizers, and network architecture. Designing products that relied on deep learning was not part of the research agenda. Deep learning became ubiquitous only after the development of Tensor-flow,[1] Keras,[s] Pytorch,[t] and other tools that standardized implementations and brought research within the reach of typical software developers. Today, deep learning is commonplace.

This was not a coincidence. A confluence of hardware advancements and development of the tooling was critical for data engineers and product designers to re-imagine products with deep learning in mind. The best tools quickly become second nature to their users. With effortless tools, engineers and product designers can focus on redesigning their systems to put privacy first. A compiler toolchain that properly encapsulates the FHE stack will be that tool.

Our work focused on a general-purpose transpiler because of its attractive abstractions and simplification it offered. Other classes of tools and infrastructure are also just as important for FHE to thrive. Domain-specific FHE compilers[4,20,21,45,47] for numerical and scientific computing can occupy a critical middle ground between black-box FHE and building your own FHE programs from scratch, as well as enable faster iteration of experiments in what is possible with FHE. Lower in the stack, improvements to cryptographic backends and hardware acceleration will improve performance without requiring end users who use an FHE compiler to completely rebuild their application. A modular compiler toolchain facilitates independent progress in each of these critical areas.

As the compiler toolchain matures, we also need standardized benchmarks, guidance for application developers around the core trade-offs of adopting FHE, and documentation in the form of tutorials, references, how-to guides, and conceptual deep dives. The road ahead is long and challenging, but a growing community of researchers, engineers, and privacy experts are essential to unlocking the potential of FHE for the world at large.

Acknowledgments. We thank Jeremy Kun, Eric Astor, Moti Yung, and Iliyan Malchev for significant contributions to this article and the entire team for their contributions to the project: S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I.M. Perera, Y. Sushko, C. Yun, A. Fitch, W. Breakell, and B. Li.

Back to Top

References

1. Abadi, M. et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

2. Archer, D.W. et al. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing amp; Applied Homomorphic Cryptography.* ACM, New York, NY, USA, 2019, 57–68.

3. Boemer, F., Kim, S., Seifu, G., de Souza, F.D.M. and Gopal, V. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. CoRR, 2021; abs/2103.16400.

4. Boemer, F., Lao, Y., Cammarota, R. and Wierzynski, C. Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM Intern. Conf. Computing Frontiers.* ACM, New York, NY, USA, 2019, 3–13.

5. Boura, C., Gama, N., Georgieva, M. and Jetchev, D. Chimera: Combining ring-LWE-based fully homomorphic encryption schemes. Cryptology ePrint Archive, Paper 2018/758; https://eprint.iacr.org/2018/758.

6. Brakerski, Z., Gentry, C. and Vaikuntanathan, V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory 6*, 3 (Jul 2014).

7. Brakerski, Z. and Vaikuntanathan, V. Efficient fully homomorphic encryption from (standard) LWE. Cryptology ePrint Archive, Paper 2011/344; https://eprint.iacr.org/2011/344.

8. Brakerski, Z. and Vaikuntanathan, V. Fully homomorphic encryption from ring-LWE and security for key dependent messages. *Advances in Cryptology—CRYPTO 2011.* P. Rogaway, ed. Springer Berlin Heidelberg, 505–524.

9. Cammarota, R. Intel HERACLES: Homomorphic encryption revolutionary accelerator with correctness for learning-oriented end-to-end solutions. In *Proceedings of the 2022 on Cloud Computing Security Workshop.* ACM, New York, NY, USA, 3

10. Carpov, S., Dubrulle, P. and Sirdey, R. Armadillo: A compilation chain for privacy preserving applications. In *Proceedings of the 3rd Intern. Workshop on Security in Cloud Computing.* ACM, New York, NY, USA, 2015, 13–19.

11. Cheon, J., Kim, A., Kim, M. and Song, Y. Homomorphic encryption for arithmetic of approximate numbers. *Advances in Cryptology.* T. Takagi and T. Peyrin, eds. Springer International Publishing, Cham, 2017, 409–437.

12. Cheon, J.H. *et al. Introduction to Homomorphic Encryption and Schemes.* Springer International Publishing, Cham, 2021, 3–28.

13. Chielle, E., Mazonka, O., Gamil, H., Tsoutsos, N.G. and Maniatakos, M. E3: A framework for compiling C++ programs with encrypted operands. Cryptology ePrint Archive, Paper 2018/1013; https://eprint.iacr.org/2018/1013.

14. Chillotti, I., Gama, N., Georgieva, M. and Izabachène, M. TFHE: Fast fully homomorphic encryption over the Torus. *J. Cryptol. 33*, 1 (Jan. 2020), 34–91

15. Chillotti, I., Gama, N., Georgieva, M. and Izabachène, M. TFHE: Fast fully homomorphic encryption library, Aug. 2016; https://tfhe.github.io/tfhe/.

16. Chillotti, I., Gama, N., Georgieva, M. and Izabachène, M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. Cryptology ePrint Archive, Paper 2016/870; https://eprint.iacr.org/2016/870.

17. Chillotti, I., Joye, M., Ligier, D., Orfila, J-B. and Tap, S. Concrete: Concrete operates on ciphertexts rapidly by extending TFHE. In *Proceedings of the 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography 15*, 2020.

18. Chillotti, I., Joye, M., Ligier, D., Orfila, J-B. and Tap, S. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. Cryptology ePrint Archive, Paper 2021/729; https://eprint.iacr.org/2021/729.

19. Coussy, P. and Morawiec, A. *High-Level Synthesis from Algorithm to Digital Circuit.* Springer Dordrecht.

20. Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K. and Musuvathi, M. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. CoRR, 2019; abs/1912.11951.

21. Dathathri, R. et al. Chet: An optimizing compiler for fully homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conf. Programming Language Design and Implementation.* ACM, New York, NY, USA, 2019, 142–156.

22. Ducas, L. and Micciancio, D. FHEW: Bootstrapping homomorphic encryption in less than a second. *Advances in Cryptology—EUROCRYPT 2015.* E. Oswald and M. Fischlin, eds. Springer Berlin Heidelberg, 2015, 617–640.

23. Fan, J. and Vercauteren, F. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144; https://eprint.iacr.org/2012/144.

24. Feldmann, A. et al. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). CoRR, 2021; abs/2109.05371.

25. Genkin, D., Shamir, A. and Tromer, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. *Advances in Cryptology—CRYPTO 2014.* J.A. Garay and R. Gennaro, eds. Springer Berlin Heidelberg, 2014, 444–461.

26. Gentry, C. A fully homomorphic encryption scheme. Ph.D. thesis. Stanford University, 2009; crypto.stanford.edu/craig.

27. Gentry, C., Sahai, A. and Waters, B. Homomorphic encryption from learning with errors: Conceptually simpler, asymptotically faster, attribute based. In *Proceedings of Advances in Cryptology-Crypto 8042*, Aug. 2013.

28. Gorantala, S. et al. A general-purpose transpiler for fully homomorphic encryption. CoRR, 2021; abs/2106.07893.

29. Halevi, S. and Shoup, V. Design and implementation of HELib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481; https://eprint.iacr.org/2020/1481.

30. Halfond, W.G.J., Viegas, J. and Orso, A. A classification of SQL-injection attacks and countermeasures. 2006.

31. Kocher, P. et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of IEEE 2019 Symp. Security and Privacy.* 1–19.

32. Kocher, P., Jaffe, J. and Jun, B. Differential power analysis. *Advances in Cryptology — CRYPTO' 99.* M. Wiener, ed. Springer Berlin Heidelberg, 1999, 388–397.

33. Lattner, C. et al. MLIR: A compiler infrastructure for the end of Moore's Law. ArXiv, 2020; abs/2002.11054.

34. Moritz Lipp, M. et al. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symp.* USENIX Assoc., Baltimore, MD, USA, Aug. 2018, 973–990.

35. Micciancio, D. and Polyakov, Y. Bootstrapping in FHEW-like cryptosystems. Cryptology ePrint Archive, Paper 2020/086, 2020; https://eprint.iacr.org/2020/086.

36. Michel, F. and Cottle, E. Optical computing for cryptography: Fully homomorphic encryption; http://bit.ly/3Jex7gH.

37. Paillier. P. Public-key cryptosystems based on composite degree residuosity classes. *Advances in Cryptology.* Springer, 1999, 223–238.

38. Peikert, C. Public-key cryptosystems from the worst-case shortest vector problem: Extended abstract. In *Proceedings of the 41st Annual ACM Symp. Theory of Computing.* ACM, New York, NY, USA, 2009, 333–342.

39. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *J ACM 56*, 6 (2009), 1–40.

40. Rivest, R.L., Shamir, A. and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

41. Rivest, R.L., Adleman, L.M. and Dertouzos, M.L. On data banks and privacy homomorphisms. *Foundations of Secure Computation.* Academic Press, 1978, 165–179.

42. Samardzic, N. et al. Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual Intern. Symp. Computer Architecture.* ACM, New York, NY USA, 2022, 173–187.

43. Smart, N.P. and Vercauteren, F. Fully homomorphic SIMD operations. Designs, Codes and Cryptography 71, 2014, 57–81.

44. Thomas, D. and Moorby, P. *The Verilog R Hardware Description Language.* Springer New York, NY, USA.

45. van Elsloo, T. Patrini, G., and Ivey-Law, H. Sealion: A framework for neural network inference on encrypted data. *CoRR*, 2019; abs/1904.12840.

46. Viand, A., Jattke, P. and Hithnawi, A. SOK: Fully homomorphic encryption compilers. CoRR, 2021; abs/2101.07078.

47. Viand, A. and Shafagh, H. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing and Applied Homomorphic Cryptography.* ACM, New York, NY, USA, 2018, 49–60.

48. Wetter, J. and Ringland, N. Understanding the impact of Apache log4j vulnerability; http://bit.ly/3kMQsvv.

Back to Top

Authors

Shruthi Gorantala is a software engineer at Google Inc. in Mountain View, CA, USA.

Rob Springer is a software engineer at Google, Inc. in Mountain View, CA, USA.

Bryant Gipson is an engineering manager at Google, Inc. in Mountain View, CA, USA.

Back to Top

Footnotes

a. Homomorphic encryption; https://bit.ly/3Hyuzsu

b. The future of crypto: IBM makes a new leap with fully homomorphic encryption; https://ibm.co/3kLDrlO

c. Microsoft SEAL 4.0; https://github.com/Microsoft/SEAL.

d. Palisade homomorphic encryption software library; https://palisade-crypto.org/documentation/

e. Tune Insight SA. EPFL–LDS. Lattigo v3 (Apr. 2022); https://github.com/tuneinsight/lattigo/

f. Alchemy; https://github.com/cpeikert/ALCHEMY

g. Hebench; https://hebench.github.io/

h. https://homomorphicencryption.org/standard/

i. LLVM; https://llvm.org/

j. Hot spot optimizations; https://research.google/pubs/pub45290/

k. FFTW; https://www.fftw.org/

l. NTL. A library for doing number theory; https://libntl.org/

m. FHE transpiler; https://github.com/google/fully-homomorphic-encryption.

n. XLS-IR semantics; https://google.github.io/xls/irsemantics/.

o. XLS; https://github.com/google/xls/.

p. XLScc; https://github.com/google/xls/tree/main/xls/contrib/xlscc.

q. Cufhe; https://github.com/vernamlab/cuFHE.

r. Introduction to FHE; https://fhe.org/talks/introduction-to-fhe-by-pascal-paillier/

s. keras; https://keras.io/

t. pytorch; https://pytorch.org/.

No entries found

Comment on this article

Signed comments submitted to this site are moderated and will appear if they are relevant to the topic and not abusive. Your comment will appear with your username if published. View our policy on comments

(Please sign in or create an ACM Web Account to access this feature.)

Create an Account

SUBMIT FOR REVIEW