A General Purpose Transpiler for Fully Homomorphic Encryption

Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson,
Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann,
Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki,
Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy,
Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson

fhe-open-source@google.com

June 14, 2021

Abstract

Fully homomorphic encryption (FHE) is an encryption scheme which enables computation on encrypted data without revealing the underlying data. While there have been many advances in the field of FHE, developing programs using FHE still requires expertise in cryptography. In this white paper, we present a fully homomorphic encryption transpiler that allows developers to convert high-level code (e.g., C++) that works on unencrypted data into high-level code that operates on encrypted data. Thus, our transpiler makes transformations possible on encrypted data.

Our transpiler builds on Google's open-source XLS SDK [1] and uses an off-the-shelf FHE library, TFHE [2], to perform low-level FHE operations. The transpiler design is modular, which means the underlying FHE library as well as the high-level input and output languages can vary. This modularity will help accelerate FHE research by providing an easy way to compare arbitrary programs in different FHE schemes side-by-side. We hope this lays the groundwork for eventual easy adoption of FHE by software developers. As a proof-of-concept, we are releasing an experimental transpiler [3] as open-source software.

1 Introduction

As cloud computing services continue to see widespread adoption, it becomes increasingly important for service providers to guarantee the security and privacy of the data of their customers. Fully homomorphic encryption (FHE) is a cryptographic technique that provides strong security guarantees since the server only ever has access to encrypted data. FHE has not seen widespread use so far, for two main reasons: FHE is still too computationally expensive to be practical, and developing FHE applications requires extensive cryptographic expertise. Fortunately, over the last few years, FHE has become much less computationally intensive, due to significant progress in hardware acceleration, efficient optimizations, and low-level implementations. However, the widespread adoption of FHE still requires available tools that allow software developers without cryptography expertise to incorporate FHE into their applications. Our work attempts to bridge this gap. We have built and open-sourced a proof-of-concept general-purpose transpiler [3] that can automatically convert a regular program that works on unencrypted data into one that performs the same operations on encrypted data.

In the remainder of this paper, the term *plaintext* refers to unencrypted data, and *ciphertext* refers to encrypted data.

2   Background

This section provides context and details on fully homomorphic encryption and transpilers for fully homomorphic encryption.

2.1   Fully Homomorphic Encryption

A *homomorphic encryption scheme* is an encryption scheme in which some operations can be performed on both plaintext or ciphertext inputs and outputs. Homomorphic encryption has a few flavors including *partially homomorphic encryption* (PHE), *fully homomorphic encryption* (FHE), and *somewhat homomorphic encryption* (SHE).

In PHE, only a subset of all possible computations can be performed on ciphertext without decryption. For example, in the additively homomorphic Paillier cryptosystem [4], the product of two ciphertexts is equivalent to the sum of two plaintexts. In the multiplicatively homomorphic RSA cryptosystem [5], the multiplication of two ciphertexts is equivalent to the multiplication of two plaintexts.

Often called the "Holy Grail" of cryptography [6], FHE is an encryption mechanism that allows both addition and multiplication (and therefore any arbitrary computation) to be performed on encrypted data [7]. Though first proposed in the 1970s [7], a theoretically feasible construction was not introduced until 2009 by Craig Gentry [8].

A SHE scheme [9] allows both addition and multiplication to be performed but only for a few computations, after which the ciphertext loses too much integrity, and can no longer be correctly decrypted.

Most modern FHE schemes rely on the "Learning with Errors" (LWE) technique [10], which relies on *noise* being added to ciphertexts. As long as the noise is sufficiently small, ciphertext can be decrypted to the correct message. During homomorphic operations, the noise in the ciphertext grows. While this effect is negligible during additions, multiplying two ciphertexts significantly increases the total amount of noise. As a result, only a fixed number of consecutive multiplications (a parameter called multiplicative depth) can be performed before decryption becomes impossible. This limitation can be circumvented using *bootstrapping*, a technique that resets the noise level of a ciphertext to a fixed lower level by homomorphically evaluating the decryption circuit (i.e., the logic that converts ciphertext into plaintext) with an encrypted secret key as input.

The first generation FHE schemes were based on the original Gentry scheme [11] and were slow, often requiring 30 mins [12] for a single multiplication. These first generation schemes convert an SHE scheme into an FHE scheme through bootstrapping.

Second generation schemes, such as BGV [13] and BFV [14], use a technique called *leveled homomorphic encryption*, which involves choosing sufficiently large parameters to allow the required computation without having to perform bootstrapping. They also introduced *SIMD-style batching* [15], an optimization technique where many messages are packed into a single ciphertext, to reduce the overall latency overhead.

Third generation schemes based on the GSW scheme [16] focus on *fast bootstrapping*, which reduces the time spent on bootstrapping by several orders of magnitude. However, fast bootstrapping does not allow SIMD-style bootstrapping and batching, which presents a tradeoff between latency and throughput when compared to second generation schemes (i.e., BGV, BFV). The TFHE scheme [17] (itself based on GSW) uses fast bootstrapping.

Developing software that uses FHE poses a unique challenge in *parameter selection*. For a specific computation, one must choose the right parameter set that avoids bootstrapping, still provides security, and ensures that the size of the ciphertext remains manageable. It also involves selecting the proper encoding scheme for the right computations. For example, binary encoding is preferable for Boolean operations (i.e., TFHE), while arithmetic encoding is preferable for arithmetic computations (i.e., BGV and BFV).

2.2   FHE Transpilers

FHE is getting closer to becoming practical, but it still requires significant expertise to incorporate it in software development. Beyond the understanding of required cryptographic parameters, prior work mostly focused on supporting low-level programming primitives (e.g., arithmetic operators or Boolean logic), so developers need additional expertise in low-level software design to use these primitives and build more complex
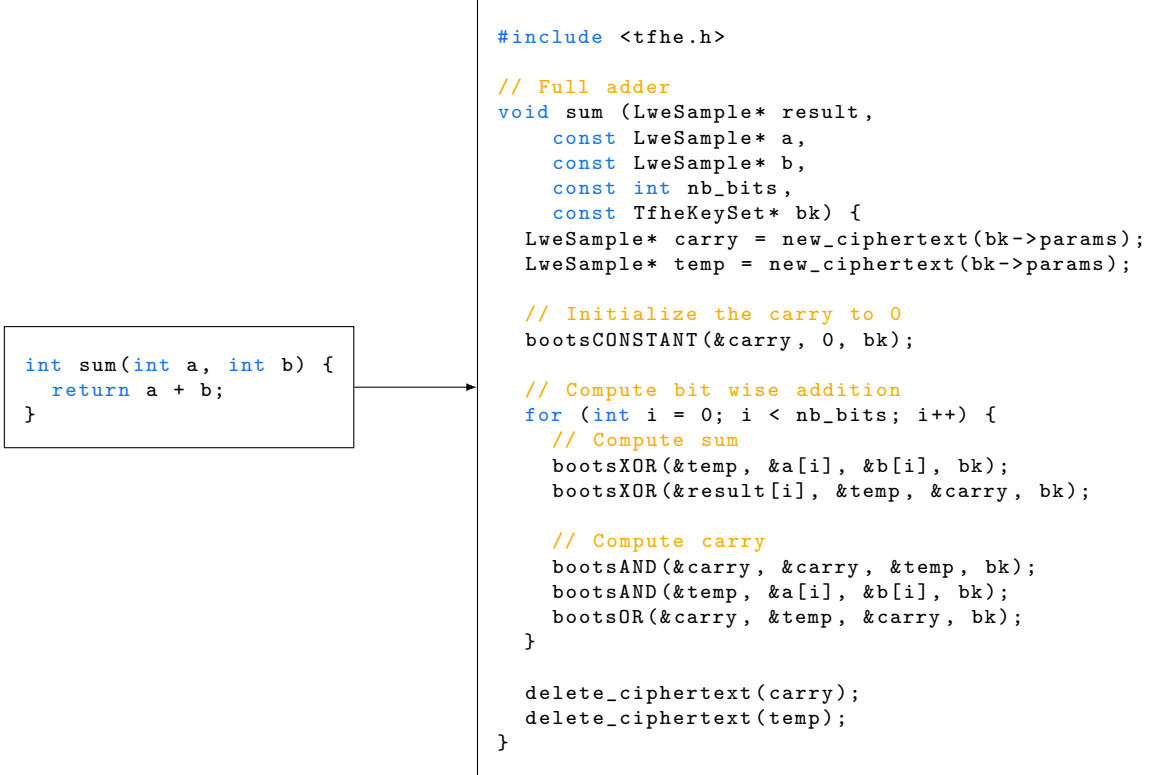
```cpp
#include <tfhe.h>

// 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 bit wise 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);
}
```

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

Figure 1: Example of a transpiled C++ program..

programs. FHE application development currently lacks a generic way for developers without cryptography expertise to write code without having to understand the underlying scheme. An FHE transpiler can bridge this gap.

A transpiler is a tool that converts one high-level language into another high-level language. An FHE transpiler takes code written in a high-level language (e.g., C++) and produces equivalent code capable of processing encrypted inputs.

As recently mentioned by Viand et al. [12], a missing component of the FHE story is a series of clean abstraction layers that separate business logic (i.e., what a developer is trying to achieve) from intermediate representation (i.e., how lower-level systems might reason or optimize around this) and optimized low-level implementation (i.e., what libraries and backends might be used to support these first two layers). Ideally, the top-most layer would accept high-level languages as input while allowing intermediate layers to remain expressive enough to represent gate operations and arithmetic operations in order to take advantage of various FHE schemes and translate among them [18], if needed. We believe that our transpiler helps fill this gap.

3   Design

Our general-purpose transpiler leverages features on the XLS toolchain and the gate level operations of TFHE to convert C++ programs into FHE-C++ programs. However, our transpiler design is modular, such that the backend FHE library, as well as the input and output languages, can be changed to suit different use cases.

3.1   External Components

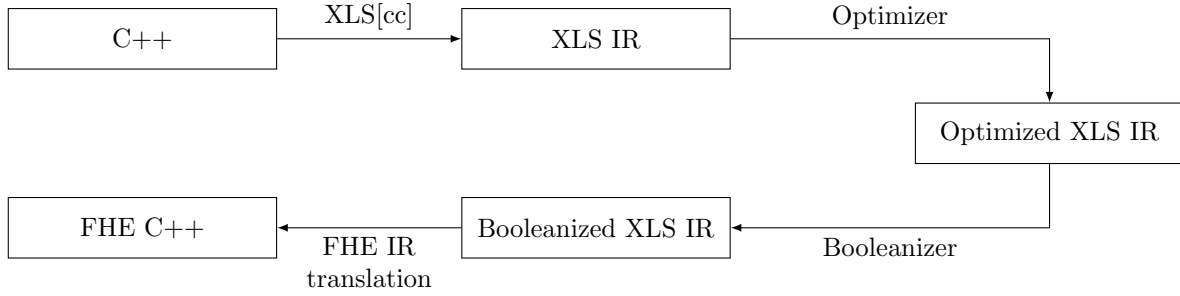Our transpiler uses two external components: XLS [1] and TFHE [2].

Figure 2: High-level overview of the transpiler invocation process.

3.1.1 XLS

XLS 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 [19]. This compilation step introduces a flexible *intermediate representation (XLS IR)* that allows for optimizations and other code transformations in the space between the high-level input language and the targeted output Verilog.

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. Fortunately, this is exactly what is needed to translate higher-level language (e.g., C++) operations into lower-level Boolean operations (i.e., gates). Our transpiler uses XLS IR as the intermediate layer between the input C++ and the TFHE library. Optimizations (e.g., reducing gate count or eliminating unneeded bits) and transformations (e.g., *Booleanification*, which is "flat-tening" an N-bit-wide operation to a series of 1-bit-wide operations) are performed on this intermediate layer.

3.1.2 TFHE

TFHE is a fast fully homomorphic encryption library over torus, proposed by Chilloti et al. [20]. It is based on GSW and its ring variants. It significantly speeds up the bootstrapping operation (less than 0.1 sec [20]) and reduces bootstrapping key size, while preserving the same security levels as long as the T(R)LWE [20] problem remains intractable.

TFHE exposes an API for gate operations. In particular, it performs a bootstrap operation for every operation on the ciphertext. As the ciphertext refreshes itself after every gate operation, it allows for unlimited computations without noise management. With the operations provided by TFHE, it is possible to represent any computation as a composition of binary inputs and logical gates. This makes it ideal for the IR layer defined in the previous section to use TFHE for transpilation.

3.2 Transpiler Invocation Stages

We have designed a general-purpose transpiler for converting standard programs into FHE programs. As a proof-of-concept, we built a transpiler that uses TFHE to convert C++ programs into FHE-C++ programs, and published it as open-source software [3]. However, our transpiler design is modular, such that the backend FHE library, as well as the input and output languages, can be changed to suit different use cases.

We now describe our transpiler by stepping through our initial open-source implementation. The sequential stages of a transpiler invocation are as follows:

- **C++ frontend:** Converts C++ input into an intermediate representation (XLS IR).

- **Optimizer:** Simplifies the IR by reducing the number of operations, replacing operations with less-intensive equivalents, reducing bit widths, and other optimizations. This is part of the XLS toolchain.

- **Booleanifier:** Replaces all multi-bit compound operations with equivalent sequences of single-bit fundamental (e.g., AND, OR, NOT) operations. This is also part of the XLS toolchain.

- **FHE IR Translator:** Constructs a C++ function invoking the TFHE library routine corresponding to each Boolean IR operation.
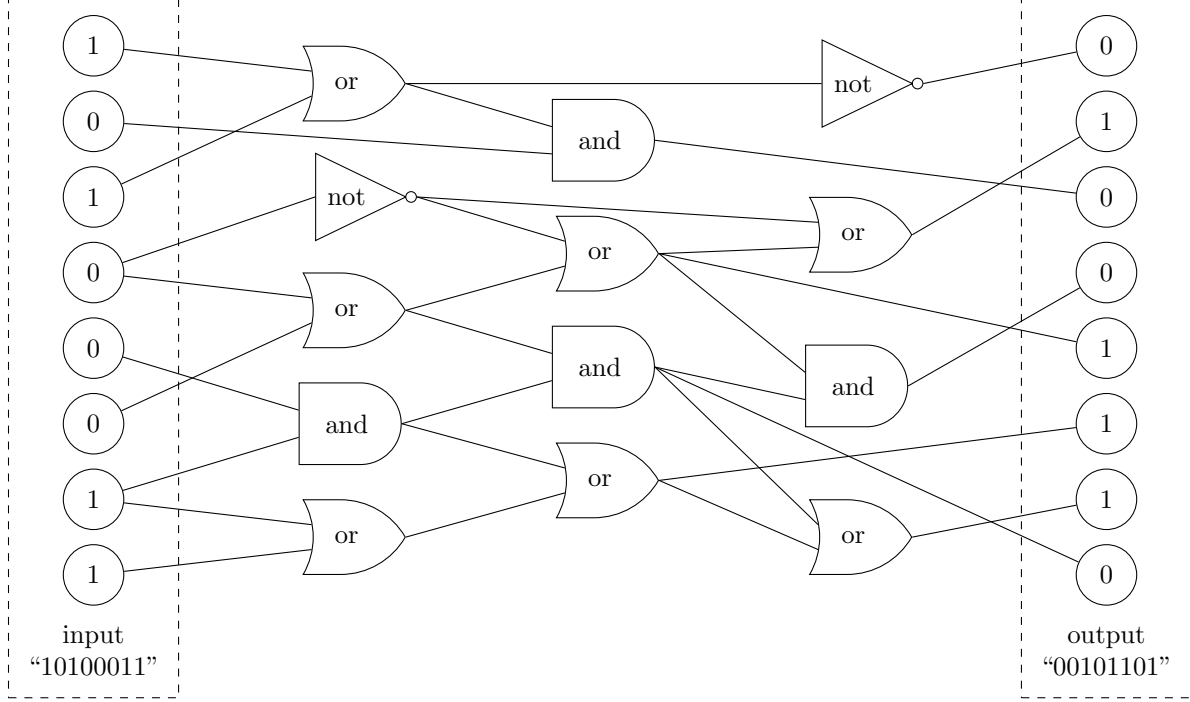
Figure 3: Visualization of a gate in XLS IR representation.

3.2.1 XLS[cc] Stage

We use XLS[cc] [21] as our frontend, compiling C++ into XLS IR.

3.2.2 Optimizer Stage

We leverage the XLS optimizer to apply various optimizations [22] to the IR produced by XLS[cc]. Our transpiler supports multiple optimization passes and the IR is Booleanified between optimization passes to take advantage of bit-level optimization opportunities.

3.2.3 Booleanifier Stage

The optimized XLS IR is translated into Boolean XLS IR, which only uses Boolean gates. This is required because TFHE only supports Boolean gates. This step is applied after each optimization pass if multiple optimization passes are requested.

3.2.4 FHE IR Translation Stage

The TFHE transpiler backend translates the Boolean XLS IR into C++ code that invokes TFHE for gate implementations. Our library also includes an FHE IR Interpreter which provides more flexibility in execution strategies, including multicore dispatch for improved performance.

This backend is deliberately constructed to be easily ported to any library that provides implementations for gates. Thus, other FHE libraries that provide the same gate-based interface can be used as drop-in replacements for TFHE. As an example, we implemented a version that uses native C++ boolean operations (without any FHE functionality), which proved useful for debugging during transpiler development.

3.3 Features

3.3.1 Modular Design

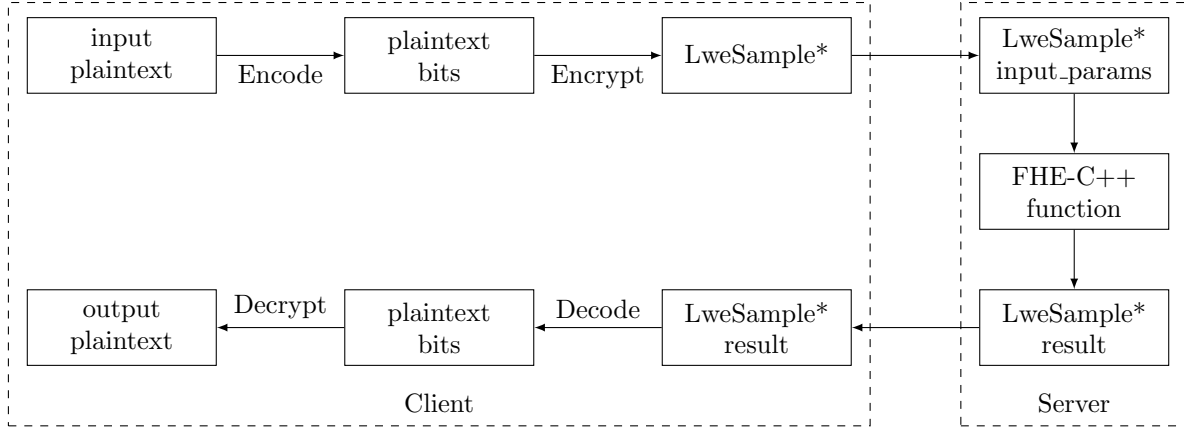Our transpiler design is modular in three ways:

Figure 4: Client server interaction.

- The <u>input code</u> can be in <u>any language that can be translated into XLS</u>. This significantly reduces the burden of transpiling existing code, since it need not be written in a fixed supported language.

- The <u>output FHE code</u> can be in <u>any language with an FHE library</u>. This reduces the requirements for interacting with transpiled FHE code, since the FHE code can be transpiled into a language that interfaces well with the rest of the code providing the computational service.

- The underlying FHE backend can be any library that exposes gates as part of its API. Our library includes classes prefixed with `Encoded`, which can be reused to ease development. This can accelerate FHE research by providing an easy way to compare arbitrary programs in different FHE schemes side-by-side.

We hope our modular transpiler design will lay the groundwork towards more research breakthroughs and eventual easy adoption of FHE by software developers.

3.3.2 TFHE Interface Utilities

To interface with a FHE-C++ program (i.e., run it on some inputs and receive the output), developers need to write code that takes some input, sends the encrypted input to the generated FHE-C++ code, and allocates memory to receive the result. In the special case of a *test bench* running the FHE-C++ program with known inputs, it also needs to encrypt the input and decrypt the result to verify it is as expected.

We help simplify these operations, providing an <u>API to encode/encrypt and decode/decrypt data while automatically managing memory allocation</u>. This includes wrappers for several standard data types (e.g., integers, arrays of integers, and strings). Our transpiler also automatically generates wrappers with the same API for user-provided data types (e.g., classes and structs). More specifically, `FheValue` and other classes with the `Fhe` prefix [23] provide a <u>simple interface for converting between native plaintext data and encrypted `LweSample*`</u> for a user that has the secret key, while also handling basic memory management. To help interface with other potential backends, including the unencrypted Boolean backend, we provide similar classes (`EncodedValue` and others with the `Encoded` prefix [24]) that convert between native plaintext data and a plaintext-bits representation.

3.3.3 Debugging Utilities

One of the obstacles for FHE adoption is the <u>inability to debug programs.</u> To help with debugging, our open-source library includes an <u>alternate backend</u> that transpiles the input to a Booleanified C++ version (without any FHE features). This uses the same computation steps as the FHE version, but <u>operates on plaintext bits.</u> This is, of course, only intended to be used for debugging purposes, as it <u>does not provide any security guarantees.</u>

4 How Best to Use the Transpiler

Our transpiler is currently experimental. We do not recommend relying on it for production use cases. This open-source release is meant to showcase the feasibility of transforming general-purpose code (written by non-experts in cryptography) to FHE programs that work on encrypted data.

We see this as an important stepping stone in privacy-protecting technologies that can be iterated upon to accelerate research innovation and build feature-rich solutions. For example, as discussed in the Modular Design section above, the transpiler can support different input and output programming languages and can be used to compare different FHE schemes. This is possible because the XLS intermediate representation is independent of both the front-end language and the underlying FHE scheme. This also decouples FHE schemes from circuits, which allows researchers and experts in respective fields to make progress independently.

4.1 FHE Programming Restrictions

Traditional imperative programming languages use data-dependent branching for control flow and optimizations. In FHE programming, computations must be *data-independent*. The execution hardware doesn't have access to the plaintext, so it cannot support branching. This implies a number of restrictions, as follows.

Variable-length loops and arrays cannot be used, and must be replaced by fixed-sized arrays and loops with a fixed number of iterations. Early returns are not useful, since the entire function body must be evaluated regardless of the input. Recursion is not supported either, since it requires data-dependent termination. Pointers are not supported as they are data-dependent. Branch-and-bound optimizations are not possible because all branches must be executed.

4.2 Threat Model

Our transpiler generates code for a server to perform computations on encrypted text. It assumes an honest-but-curious adversary, whereby the server can be trusted to do the expected computation and knows the type of the data, but cannot be allowed to decrypt or view the data itself. The data type (e.g., int, string, class) is known to the server. The length of the data (e.g., the size of an input array) needs to be explicitly protected on the client-side by enforcing a fixed maximum input length (using padding).

The generated code does not protect against an adversarial server's malicious manipulation of the data. This is inherent in all FHE schemes, given that the goal of FHE is to allow the server to compute functions on the data, but does not provide verification of what function was computed.

4.3 Security Considerations

The TFHE scheme [17] bases its security on the torus variant of the (R)LWE problems. In cryptography, the reputation and credibility of a cryptosystem is typically established through work of cryptanalysis experts who have unsuccessfully tried to break such a scheme over multiple years. In contrast, the TFHE scheme is new and has not enjoyed as much public scrutiny as widely-deployed cryptographic primitives. The same paper that introduced TFHE [17] presented a thorough cryptanalysis assessment, but the cryptography community may still eventually find more efficient algorithms that break the underlying premises of TFHE, which would impact the security estimates of the parameters used by the transpiler tool.

5 Related Work

Our FHE Transpiler's main novelty is that it generates debuggable high-level language output. Another primary contribution is the introduction of XLS as an expressive intermediate representation that is independent from both the front-end language and the underlying FHE scheme. This allows interoperability between languages and even FHE schemes. By decoupling the FHE scheme from the implementation circuits, researchers and experts in respective fields are free to make progress independently. Finally, it provides a framework to compare new FHE schemes and cryptographic optimizations uniformly on different programs.

However, the concept of automatically converting programming language source code into FHE-related implementations is not new. The Armadillo framework [25] (currently called the Cingulata toolchain), for instance, is also based on the TFHE scheme and converts C++ into Boolean circuits, performs gate operations, and converts the Boolean operations to FHE binary operations. In comparison to our work, Cingulata currently does not convert the source code back to any C++ source code. Furthermore, our transpiler supports higher-level operations such as SUM, MUL, and DIV in addition to gate operations. Converting the IR to FHE-C++ supports extensibility to a wider variety of FHE libraries and helps in debugging.

Another similar tool, also based on TFHE, is Encrypt-Everything-Everywhere (E3) [26]. It 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. E3 converts the compiled netlist into C++ functions that can be called directly from the program. In comparison, our transpiler works on pre-existing programs in supported high-level languages, automatically generating the transpiled FHE-C++ and relevant libraries to encode the data types involved in the program (including structs and classes) and does not need an explicit configuration file.

A few other similar tools, based on different (non-TFHE) underlying FHE schemes, have been proposed in the literature. For example, ALCHEMY [27], Marble [28] and RAMPARTS [29] 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 the specific topic of building FHE compiler tools for specific workloads. For example, nGraph-HE [30], SEALion [31], and CHET [32] all intend to produce efficient and FHE-friendly code for certain machine learning workloads.

6 Future Directions

A natural direction for future work on a generic FHE transpiler like ours is to improve execution times. We believe that the XLS intermediate representation can help with reasoning about, and implementing, various optimizations. For instance, if programmable bootstrapping [33] is supported in the underlying FHE scheme, this would allow optimizations using high-performance versions of arbitrary univariate functions.

The transpiler currently uses bitwise operations, so all arithmetic operations are converted into many single-bit Boolean gates, making them quite slow. MUL and DIV/MOD of two ciphertext values are especially expensive. Support for native arithmetic operations and SIMD-style packing with CHIMERA [18] would help speed up arithmetic operations and improve throughput.

The interpreter in the FHE IR Translation stage provides more flexibility in execution strategies, including multicore dispatch for improved performance. We intend to add additional optimizations that will be especially useful for heterogeneous compute environments, such as mixed CPU/GPU execution.

Automatic parameter selection has remained a challenging problem in the field of FHE because of the unknown circuits for programs. As our transpiler generates circuits for arbitrary programs, it could be augmented for automatic parameter selection at transpilation time.

References

[1] Google XLS. https://github.com/google/xls.

[2] TFHE: Fast Fully Homomorphic Encryption over the Torus. https://tfhe.github.io/tfhe.

[3] Google FHE Transpiler. https://github.com/google/fully-homomorphic-encryption/tree/main/transpiler.

[4] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology*, pages 223–238. Springer, 1999.

[5] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[6] Dalia Tourky, Mohamed ElKawkagy, and Arabi Keshk. Homomorphic Encryption the "Holy Grail" of Cryptography. In *International Conference on Computer and Communications*, pages 196–201. IEEE, 2016.

[7] Ronald Linn Rivest, Leonard Max Adleman, and Michael Leonidas Dertouzos. On Data Banks and Privacy Homomorphisms. In *Foundations of Secure Computation*, pages 165–179. Academic Press, 1978.

[8] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[9] Homomorphic Encryption. `https://en.wikipedia.org/wiki/Homomorphic_encryption`.

[10] Pedro Silveira Pisa, Michel Abdalla, and Otto Carlos Muniz Bandeira Duarte. Somewhat Homomorphic Encryption Scheme for Arithmetic Operations on Large Integers. In *Global Information Infrastructure and Networking Symposium*, pages 1–8. IEEE, 2012.

[11] Craig Gentry and Shai Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–148. Springer, 2011.

[12] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully Homomorphic Encryption Compilers. In *Symposium on Security and Privacy*. IEEE, 2021.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory*, 6(3):1–36, 2014.

[14] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

[15] Nigel Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.

[16] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology*, pages 75–92. Springer, 2013.

[17] 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.

[18] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. Cryptology ePrint Archive, Report 2018/758, 2018. `https://eprint.iacr.org/2018/758`.

[19] Verilog. `https://en.wikipedia.org/wiki/Verilog`.

[20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology*, pages 3–33. Springer, 2016.

[21] Google XLS[cc]. `https://github.com/google/xls/tree/main/xls/contrib/xlscc`.

[22] Google XLS Optimizations. `https://google.github.io/xls/optimizations`.

[23] Google FHE Data. `https://github.com/google/fully-homomorphic-encryption/blob/main/transpiler/data/fhe_data.h`.

[24] Google FHE Boolean Data. `https://github.com/google/fully-homomorphic-encryption/blob/main/transpiler/data/boolean_data.h`.

[25] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *International Workshop on Security in Cloud Computing*, page 13–19. ACM, 2015.

[26] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A Framework for Compiling C++ Programs with Encrypted Operands. Cryptology ePrint Archive, Report 2018/1013, 2018. `https://eprint.iacr.org/2018/1013`.

[27] ALCHEMY. `https://github.com/cpeikert/ALCHEMY`.

[28] Alexander Viand and Hossein Shafagh. Marble: Making Fully Homomorphic Encryption Accessible to All. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, page 49–60. ACM, 2018.

[29] Ramparts. `https://galois.com/project/ramparts`.

[30] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In *International Conference on Computing Frontiers*, page 3–13. ACM, 2019.

[31] van Elsloo, Tim and Patrini, Giorgio and Ivey-Law, Hamish. SEALion: A framework for neural network inference on encrypted data. arXiv Preprint arXiv:1904.12840, 2019. `https://arxiv.org/abs/1904.12840`.

[32] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *SIGPLAN Conference on Programming Language Design and Implementation*, page 142–156. ACM, 2019.

[33] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. Cryptology ePrint Archive, Report 2021/091, 2021. `https://eprint.iacr.org/2021/091`.