ILA: Correctness via Type Checking for Fully Homomorphic Encryption

Anonymous Author(s)

Abstract

RLWE-based Fully Homomorphic Encryption (FHE) schemes add some small *noise* to the message during encryption. The noise accumulates with each homomorphic operation. When the noise exceeds a critical value, the FHE circuit produces an incorrect output. This makes developing FHE applications quite subtle, as one must closely track the noise to ensure correctness. However, existing libraries and compilers offer limited support to statically track the noise. Additionally, FHE circuits are also plagued by wraparound errors that are common in finite modulus arithmetic. These two limitations of existing compilers and libraries make FHE applications too difficult to develop with confidence.

In this work, we present a *correctness-oriented* IR, Intermediate Language for Arithmetic circuits, for type-checking circuits intended for homomorphic evaluation. Our IR is backed by a type system that tracks low-level quantitative bounds (e.g., ciphertext noise) without using the secret key. Using our type system, we identify and prove a strong *functional correctness* criterion for ILA circuits. Additionally, we have designed ILA to be maximally general: our core type system does not directly assume a particular FHE scheme, but instead axiomitizes a *model* of FHE. We instantiate this model with the exact FHE schemes (BGV, BFV and TFHE), and obtain functional correctness for free.

We implement a concrete type checker ILA, parameterized by the noise estimators for three popular FHE libraries (OpenFHE, SEAL and TFHE-rs). We also use the type checker to infer the optimal placement of modulus switching, a common noise management operation. Evaluation shows that ILA type checker is sound (always detects noise overflows), practical (noise estimates are tight) and efficient.

1 Introduction

Unlike traditional encryption schemes, a *fully homomorphic encryption* (FHE) enables computations on encrypted data. Popular FHE schemes—such as BGV [7], CKKS [9], TFHE [11]—that are based on Ring Learning With Errors (RLWE) adds some small *noise* to the message to make the encryption non-deterministic (and thus secure). This noise accumulates with each homomorphic operation. A crucial invariant that needs to be preserved for the successful decryption is that the noise must not exceed a *critical value*; otherwise, the decryption will not represent the accurate value. Moreover, noise overflows are attack vectors for secret key recovery attacks [19]. Noise management operations such as *modulus switching* and *programmable bootstrapping* (PBS) reduce noise but are computationally expensive, requiring judicious use. Additionally, all these schemes operate in finite rings (e.g., $\frac{Z_q[x]}{x^d-1}$); a value overflow could potentially corrupt the output. Thus, FHE application developers need tools that guarantee correct evaluation.

Unfortunately, none of the existing FHE compilers [10, 14, 15, 18] or libraries—such as OpenFHE [5], SEAL [22], HELib [16], TFHE-rs [24], Lattigo [4]— guarantee correct FHE evaluation in the presence of overflows in noise or value. Indeed, to avoid incorrect outputs, they either rely on programmer expertise or require one to simulate many runs of the FHE circuit, which carries a high computational cost and thus harms iterative development. Additionally, *no* realistic FHE library to date can simultaneously reason about overflows in both RLWE-induced noise and overflows in the underlying encrypted values.

Thus, none of the existing libraries offer a generic yet robust correctness reasoning framework that statically and automatically detects overflows.

In response, we present ILA, a *correctness-oriented* intermediate representation (IR) for *statically typechecking* a *family* of RLWE-based circuits intended for homomorphic evaluation. ILA is backed by a type system that tracks low-level quantitative bounds (e.g., ciphertext noise, value ranges) without using the secret key. Using our type system, we identify and prove a strong *functional correctness* criterion for ILA circuits: if C is well-typed, then homomorphically evaluating C, and then decrypting, is equivalent to running a cleartext version C_{clear} on its decrypted inputs. In essence, this correctness criterion guarantees that FHE does not introduce any additional functional bugs, relative to the original, non-homomorphic circuit.

A crucial insight for ILA is that RLWE-based FHE schemes share the same high-level characteristics for homomorphic and noise management operations. Thus, we have designed ILA to be maximally general: our core type system does not directly assume a particular FHE scheme, but instead axiomatizes a *model* of FHE. Thus, ILA represents a family of FHE models. In turn, we are able to instantiate this model with the widely used non-approximate schemes: BGV, BFV and TFHE.

Importantly, all ILA model instantiations inherit functional correctness for free. This is notable as the correctness guarantees can be used to validate noise-preserving transformations. For instance, ILA's BGV instantiation validates an optimization involving modulus switching. Similarly, ILA's TFHE instantiation decouples standard homomorphic and PBS operations; this enables deploying faster but correct TFHE circuits that are PBS-free.

We do not intend ILA to be used directly, but instead to provide a layer of correctness validation on top of an existing pipeline for FHE. To demonstrate this use case, we implement an extensible ILA compiler with static type checkers parameterized by the scheme. ILA compiler supports multiple FHE backends (e.g., SEAL, OpenFHE and TFHE-rs) for BGV, BFV and TFHE schemes. Using the type checker, one can obtain, for free, an FHE circuit that is formally verified for functional correctness.

Our evaluation shows that the overflow analysis employed in the type checker is sound (no false negatives) and practical (fewer false

positives). More importantly, the entire analysis is static: neither a secret key nor circuit execution is required. This brings down the application debugging costs by enabling offline development.

Finally, we present a novel insight on the placement of *modulus switching* (MS), a common noise managing optimization for FHE circuits. Developers use MS to improve the multiplicative depth (the longest sequence of ciphertext multiplication) of the circuit. However, one cannot simply annotate the entire circuit with MS operations as they are expensive. Moreover, incorrect placement could counteract any potential multiplicative depth gains; worse the circuit may even fail to run as modulus-switched ciphertexts can only operate with equivalently switched ciphertexts.

Our MS inference procedure is guided by the type checker: it not only picks placement that maximizes the remaining noise budget but also validates the correctness of the resulting circuit.

Contributions. In summary, we:

- present ILA, a generic and formal intermediate language and type system that embeds the notion of overflow(s) for FHE circuits in a scheme-agnostic manner;
- formally specify and prove a (functional) correctness theorem for ILA, which guarantees that well-typed circuits evaluate correctly;
- instantiate ILA with BGV, BFV and TFHE, the three popular absolute schemes, demonstrating the generality of ILA. Notably, TFHE operations are decoupled from the expensive programmable bootstrapping enabling users to deploy not only functionally correct but also runtime-efficient FHE circuits;
- transformation validation of our novel modulus switching inference algorithm;
- implement ILA compiler targeting three backends and a static type checker that detects both noise and value overflows for all the three schemes; importantly, the type checker does not require a secret key to estimate the noise in the ciphertext nor does it run the expensive FHE circuit; and
- demonstrates that ILA is both expressive and effective with tight noise bounds; ILA's static approach at detecting overflows outperforms the state-of-the-art dynamic approach by orders of magnitude.

We make our implementation and evaluation case studies available as an anonymous GitHub repository[1].

2 Background

A **R**ing **L**earning **W**ith **E**rrors-based encryption scheme contains three steps: First, a probabilistic keygen algorithm that on input λ, generates a secret key sk and a public key pk. Second, a probabilistic encryption algorithm generates a ciphertext for a given pk and a message m taken from the message space msg. The encryption can be further seen as a two-step process, the first step encodes a message m to an element p of the *plaintext* space. The second step encrypts the plaintext p and adds a random noise for security generating an element ct of the *ciphertext* space. Finally, a deterministic decryption algorithm takes a ciphertext ct and sk and emits a plaintext message p followed by a decoding step that emits the corresponding message m.

[1] https://github.com/anon-ila/ila

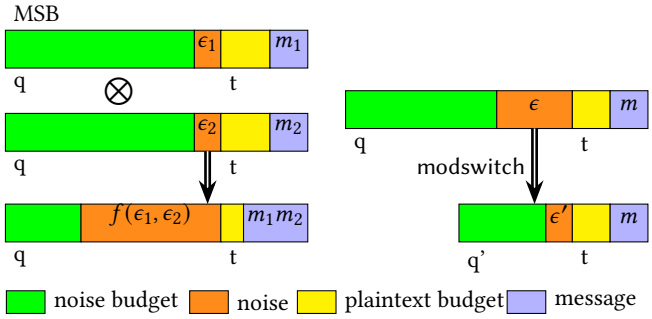

Figure 1: Noise changes during binary and unary homomorphic operations. The noise grows multiplicatively with ciphertext multiplication (left). Modulus switching reduces both noise and the remaining noise budget (right).

In an RLWE-based scheme, the plaintext space is a ring. The plaintext space encodes a message (or cleartext) space. The message space is usually $\mathbb{Z}_t$ (the ring of integers modulo t) or $\mathbb{Z}_t^d$ (d tuples of $\mathbb{Z}_t$). The plaintext space is usually the ring $R_t = \frac{\mathbb{Z}_t[x]}{(x^d+1)}$. The *plaintext modulus* t is a positive integer and d is usually a power of 2. Elements of this ring may be seen as polynomials of degree up to $d-1$ with coefficients in $\mathbb{Z}_t$. Addition and multiplication are usual polynomials addition and multiplication modulo $(t, (x^d+1))$. The ciphertext space is R_q^{n+1}, where the *coefficient modulus* q is a large positive integer. Elements of this space are $n+1$ tuples with each component a polynomial from R_q. For simplicity, the reader can assume $n = 1$.

Absolute RLWE FHE Schemes. BGV, BFV and TFHE are absolute RLWE FHE schemes: the decrypted output of an encrypted computation is the same as the computation on the underlying message. By contrast, CKKS is an approximate scheme: the decrypted output contains a small error. In this work, we focus on absolute schemes. All of them operate on integers and support (binary) arithmetic homomorphic operations, $\otimes$ and $\oplus$ for homomorphic multiplication and addition, respectively, and $\times$ for scalar multiplication of a ciphertext. BGV also supports modswitch, a modulus switching (unary) operation that reduces the noise in a ciphertext. TFHE supports three different types of ciphertexts—LWE, RLWE and RGSW—and different multiplication operators—$\otimes$, $\boxtimes$, and $\boxdot$—indexed by the ciphertext type. This makes the scheme more challenging.

Noise changes. Homomorphic operations change the ciphertext noise. Figure 1 illustrates the noise changes for multiplication and modulus switching. A ciphertext is represented as a polynomial. The length (number of bits) of each coefficient is given by coefficient modulus q. The message m_1 can grow until the size of the plaintext modulus is given by t bits. Every ciphertext has a current noise ϵ_1 that can grow until $(q-t)$ bits. If it grows beyond, then an overflow occurs.

In a ciphertext multiplication, the noise growth is proportional to the product of noises in the individual ciphertexts. Modulus switching refreshes the noise—it reduces the size of the coefficient modulus by switching to a different level. In doing so, it reduces

both the noise and noise budget leaving the message size and plaintext modulus unchanged. However, it reduces noise more than the noise budget. This relative gain in the noise budget enables more homomorphic operations.

Programmable Bootstrapping (PBS). Like modulus switching, PBS is also a noise management operation, however, it not only refreshes more noise budget but is also computationally more expensive. TFHE uses PBS to apply a lookup table (LUT) to LWE ciphertext(s). Technically, it is a composition of modulus switching, blind rotation and sample extraction operations. For PBS to succeed, there should be a non-zero noise budget. Since multiplication is expensive, TFHE tightly couples it with PBS. For other operations, PBS is added after a fixed number of operations.

3 Overview

ILA is a family of intermediate languages for developing FHE applications. At a high level, ILA is a standard imperative language with abstract homomorphic operations and types parameterized by the FHE scheme. Our key insight is driven by the observation that all RLWE-based FHE schemes share common characteristics such as the scheme parameters selection, cipher and plaintext construction as well as the propagation of noise through the circuit. As a part of parameter selection, all schemes choose a set of keys (public, private, and evaluation), coefficient modulus q, plaintext modulus t and a poly modulus degree d. Ciphertext and plaintexts are constructed as elements of finite rings involving the above chosen parameters.

The core language provides a common semantic framework for statically reasoning about the correctness of RLWE-based FHE schemes. Each scheme instantiates ILA with scheme-specific operations. For example, arithmetic operations on ciphertexts are common in all schemes, whereas noise management operations are only selectively supported. Both BGV and TFHE support modulus switching but only TFHE supports PBS. As described in Section 2, TFHE also supports multiple operations on different types of ciphertexts having distinct properties.

For an instantiation to be valid, it has to meet certain requirements (described in Section 4). A valid instantiation inherits, for free, all the correctness guaranteed offered by ILA. In Section 5, we describe $\text{ILA}_{(bgv)}$, $\text{ILA}_{(bfv)}$ and $\text{ILA}_{(tfhe)}$, instantiations for BGV, BFV and TFHE schemes, respectively. Furthermore, we prove that they are valid.

Figure 2 shows the implementation of private set intersection (PSI) in ILA [8]. The goal of PSI is to find if the intersection between two encrypted sets, A and B, (owned by different parties but encrypted using the public key owned by one of the parties) is non-empty. This is computed by subtracting each element of A from each element of B, and accumulating the difference in result. However, a naive implementation will be insecure as the decrypted result might leak more information about the other set. In the secure version, the computations are mixed with randomized data.

In the program, sets A and B are represented as vectors; A[i] denotes the element at index i. Both A and B are initialized encrypted vectors (e.g., A = cipher_init[1, 3, 36]). Elements of A and B are iterated using while loops and the subtraction is computed using the combination of multiplication and addition. PSI is non-empty if the

```
1  # Private Set Intersection between encrypted sets A and B
2  result = 1
3  n_unit = -1
4
5  for j = len_A downto 0: # begin outer loop
6      t1 = B[j]            # index into set B
7      t2 = n_unit ⊗ t1     # compute -B[j]
8      t3 = random_data[j]
9      t4 = result ⊗ t3
10     i = len_B
11     while i = len_B downto 0: # begin inner loop
12         t5 = A[i]          # index into set A
13         t6 = t5 ⊕ t2       # t6 = A[i] - B[j]
14         t7 = result ⊗ t6
15         result = t7 ⊗ t4 # 0 if PSI is non-empty
```

Figure 2: ILA program for private set intersection (PSI) over encrypted sets A and B.. *The sets are represented as vectors; operators ⊕ and ⊗ represent homomorphic addition and multiplication.*

final value of the decrypted result is zero. Note that the operations on A and B in lines 9 and 13-15 are homomorphic: ⊗ and ⊕ correspond to homomorphic multiplication and addition, respectively. Each homomorphic operation adds noise, and thus the noise gets accumulated in the result. The amount of noise is dependent on the type as well as the number of homomorphic operations: multiplication adds more noise than any other operations. If a ciphertext undergoes a sequence of multiplications, measured in terms of *multiplicative depth*, then it quickly exhausts the noise budget leading to incorrect output due to the noise overflow. Additionally, each homomorphic operation could potentially increase the message size; if this increases beyond the threshold value (plaintext modulus chosen during the initial setup), then the message wraps around leading to a value overflow.

Thus, even though the PSI program is logically correct, the output is not guaranteed to be valid for all inputs. Crucially, it depends on the depth of the circuit (in this case, loop iterations determined by len_A and len_B) and the actual inputs. For example, in SEAL, it fails to execute correctly for larger sets (e.g., len_A is greater than 4) due to noise overflows. In OpenFHE and TFHE-rs, it fails when the input values are large due to value overflows. It is thus important to know when the output is valid.

ILA's approach is to statically track the overflows using an expressive type system. Notably, the type system embeds an abstract notion of bounds for the type of polynomials used to in the construction of ciphertext and plaintexts. It tracks the effect of homomorphic operations on these bounds. The type system constrains the overflows by ensuring that they are always below the threshold values (chosen during the setup). For example, if the product of two ciphertexts with types cipher α^1_{cipher} and cipher α^2_{cipher} has the type cipher α_{cipher}, then the type system computes that $\alpha_{\text{cipher}} = f(\alpha^1_{\text{cipher}}, \alpha^2_{\text{cipher}})$ and checks if α_{cipher} is less than the threshold values.

Since ILA is parametric in the FHE scheme, each scheme instantiates the abstract bounds with precise information. For example, $\text{ILA}_{(bgv)}$ instantiates the ciphertext type cipher α_{cipher} with cipher $\langle \text{inf}, \text{sup}, \epsilon, \omega \rangle$: intuitively, the ciphertext at level ω has the current noise ϵ and the underlying messages lies in the interval

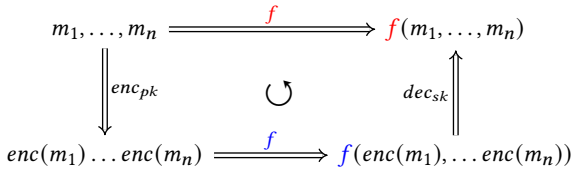

Figure 3: Functional correctness via commutativity in ILA. *Evaluation on top (in red) is over cleartext messages, while evaluation on bottom (in blue) is homomorphic over ciphertexts.*

$[\inf, \sup]$. Multiplying two ciphertexts with bounds cipher $\langle \inf_i, \sup_i, \epsilon_i, \omega \rangle$ yields the new bound cipher $\langle \inf, \sup, \epsilon, \omega \rangle$ where $\epsilon = f(\epsilon_1, \epsilon_2)$ and $[\inf, \sup] = [\min(\inf_1, \inf_2), \max(\sup_1, \sup_2)]$. Our formalism leaves f as an abstract function with the only requirement that it is monotonic. This enables one to instantiate ILA in multiple ways by picking the right noise estimator from the existing literature [11–13, 17]. Note that our work itself does not introduce any new noise estimation technique; rather our plug-and-play estimator framework is conducive to user defined estimations. For example, $f(\epsilon_1, \epsilon_2) = \epsilon_1 \times \epsilon_2$ is a worst-case estimator. We discuss noise estimators later in Section 7.

Functional correctness. Suppose the above program passes the type checker, and the circuit executes. Also suppose that the underlying messages, that is, $\text{decrypt}_{\text{sk}}(\text{A})$ and $\text{decrypt}_{\text{sk}}(\text{B})$ are $[m_1^A, m]$ and $[m, m_2^B]$, respectively. If the type checker is sound, then we should obtain that $\text{decrypt}_{\text{sk}}(\text{result}) = 0$ (since intersection is non-empty).

Generalizing the above example, we formalize the functional correctness as a relational property of a pair of executions. The first execution runs the FHE circuit as intended, whereas the second execution runs the *same* circuit in *cleartext*, with plaintext inputs that correspond to the ciphertext ones. We then require that the two executions must match; that is, the first output decrypts to the second one. In other words, the diagram commutes, as shown in Figure 3. Theorem 4.4 from Section 4 states that well-typed ILA programs are functionally correct (given that the underlying FHE operations are, with appropriate noise bounds).

Noise Management. Going back to the running example, if ILA's type checker rejects the program, we cannot guarantee that result will decrypt to 0. Indeed, the program fails to typecheck in BGV scheme if len_A is 4. In this case, one has to use modswitch, a noise refreshing operation (or PBS in TFHE), to gain additional noise budget. Using the noise budget estimates provided by ILA, line 9 could be modified to $\text{t4} = \text{result} \otimes \text{t4}$ to gain additional noise budget. If it still fails, depending on the amount of the noise budget either the following operations in lines 13-15 could be switched or line 9 could be further switched to a lower level in the modulus chain. This process repeats until the type checker accepts or the chain gets fully exhausted. In the latter case, the circuit runs without functional correctness guarantees.

The above iterative process shows that ILA's type checker could be used as a layer for validating noise refreshing transformations. In Section 6, we describe an inference algorithm that exploits this property to help with the optimal placement of modswitch operations in loop-free programs. More generally, ILA provides a validation layer for bounds-preserving transformations. In the future, we plan to use the validation layer as a foundation for building correct-by-construction circuit optimizations.

Formal ILA Model. To summarize, we define *ILA model* using an abstract core language. Values in the model are categorized into three sorts—message, encoded plaintext, and ciphertext—and each sort is associated with a corresponding set of abstract bounds, such as ciphertext noise and value intervals. Crucially, the bounds set forms a partial order, and operations over values manipulate these bounds. When the bounds exceed the scheme-specific special elements of the poset, the circuit fails.

We prove that the abstract FHE model enforces functional correctness. Interestingly, instantiating the abstract model with a suitable FHE scheme gives functional correctness for free. This is significant, as any correct instantiation of ILA now enjoys the functional correctness property. Concretely, we instantiate ILA with absolute FHE schemes to obtain $\text{ILA}_{(bgv)}$, $\text{ILA}_{(bfv)}$ and $\text{ILA}_{(tfhe)}$ and furthermore prove that they are correct instantiations of ILA. Thus programs in instantiated models inherit the functional correctness property. This demonstrates the model's general applicability to RLWE-based FHE schemes.

4 ILA: A formal model for FHE

We now present ILA, our core type system for FHE. ILA is parameterized by a *model*, which encapsulates all necessary semantic details for homomorphic evaluation. Given an arbitrary model, we then construct a type system for homomorphic circuits and prove that all circuits constructible under our type system are functionally equivalent to their non-homomorphic counterparts.

4.1 ILA Models

To use ILA with a particular FHE scheme (e.g., BGV) one needs to define a *model*, which serves as the semantic interface between the FHE scheme and ILA's type system.

Definition 4.1. *An ILA model* M *is given by the following data:*

- *Two sets* $\mathcal{PP}$ *and* $\mathcal{SP}$ *for* public *and* secret *parameters, respectively, with a mapping* $\text{topub} : \mathcal{SP} \to \mathcal{PP}$*;*
- *Sets* $[\![s]\!]$ *for each sort* $s \in \mathcal{S} = \{\text{msg}, \text{plain}, \text{cipher}\}$ *(meaning messages, (encoded) plaintexts, and ciphertexts, respectively);*
- *Sets* B_s *of* bounds *for each sort* $s \in \mathcal{S}$*, partially ordered by* $\leq_s$*;*
- *Mappings* $|\cdot|^{\text{pp}}_{\text{msg}} : [\![\text{msg}]\!] \to B_{\text{msg}}$ *and* $|\cdot|^{\text{pp}}_{\text{plain}} : [\![\text{plain}]\!] \to B_{\text{plain}}$*, where* $\text{pp} \in \mathcal{PP}$*;*
- *A mapping* $|\cdot|^{\text{sp}}_{\text{cipher}} : [\![\text{cipher}]\!] \to B_{\text{cipher}}$*, where* $\text{sp} \in \mathcal{SP}$*;*
- *A mapping* $\text{interp}^{\text{pp}}_{\text{plain}} : [\![\text{plain}]\!] \to [\![\text{msg}]\!]$*, where* $\text{pp} \in \mathcal{PP}$*;*
- *A mapping* $\text{interp}^{\text{sp}}_{\text{cipher}} : [\![\text{cipher}]\!] \to [\![\text{msg}]\!]$*, where* $\text{sp} \in \mathcal{SP}$*;*
- *A set of operations* op *of arity* $\overrightarrow{s_i} \to s$*, where each* $s, s_i \in \mathcal{S}$*, equipped with three mappings:*
  - $[\![\text{op}]\!] : \overrightarrow{[\![s_i]\!]} \to [\![s]\!]$*;*
  - *A partial mapping* $[\![\text{op}]\!]^{\text{pp}}_{\text{bnd}} : \overrightarrow{B_{s_i}} \rightharpoonup B_s$*, where* $\text{pp} \in \mathcal{PP}$*;*
  - *A mapping on messages* $[\![\text{op}]\!]_{\text{msg}} : \overrightarrow{[\![\text{msg}]\!]} \to [\![\text{msg}]\!]$*, with the appropriate input arity.*

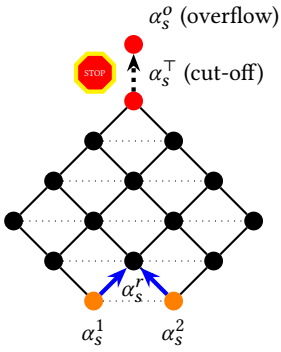

Figure 4: A partially-ordered bounds set B_s (indexed by sort s). For any given two points α_s^1 and α_s^2, the operator bounds mapping $[\![\text{op}]\!]^{\text{pp}}_{\text{bnd}}$ computes the result bounds α_s^r. The operation is well-defined iff $\alpha_s^r \leq \alpha_s^\top$ where $\alpha_s^\top$ is the cut-off for the output to be valid.

We now walk through Definition 4.1. Each ILA model has a set of public and secret parameters, which refer directly to the public and secret keys of the FHE scheme in question. Then, each ILA model defines a set of messages, encoded plaintexts, and ciphertexts, captured through giving semantics $[\![\cdot]\!]$ to the three sorts $\mathcal{S} = \{\text{msg}, \text{plain}, \text{cipher}\}$, respectively. To model quantitative concerns (e.g., ciphertext noise and integer overflow in the plaintext modulus), each sort s also comes equipped with a partially ordered set of *bounds*. For RLWE-based schemes, the bounds for messages and encoded plaintexts will track integer norms to prevent overflow in the modulus, while the bounds for ciphertexts will additionally track encryption-induced noise. We assume bounds for messages and encoded plaintexts can be computed using the public key via $|\cdot|^{\text{pp}}_{\text{msg}}$ and $|\cdot|^{\text{pp}}_{\text{plain}}$, while computing bounds for ciphertexts via $|\cdot|^{\text{sp}}_{\text{cipher}}$ needs the secret key .

Next, we have *interpretation* functions $\text{interp}^{\text{pp}}_{\text{plain}}$ and $\text{interp}^{\text{sp}}_{\text{cipher}}$, mapping their respective sorts to messages. The interpretation function for plain takes the public parameters as an argument, while the interpretation function for cipher takes the secret parameters. Interpreting encoded plaintexts corresponds to *decoding*, and requires the public parameters, while interpreting ciphertexts corresponds to *decrypting*, and requires the secret parameters. For notational convenience, we define $\text{interp}^{\text{pp}}_{\text{msg}}$ to be the identity function, and for $s \in \{\text{msg}, \text{plain}\}$, we define $\text{interp}^{\text{sp}}_s$ to be $\text{interp}^{\text{topub(sp)}}_s$. We omit the sort argument to $|\cdot|$ and interp when it is clear from context.

Finally, we have a set of operations of varying arity, such as binary homomorphic addition and multiplication, operations on messages/encoded plaintexts, and unary *noise management* operations, such as modulus switching. Other than its native semantics $[\![\text{op}]\!]$, each operation comes equipped with a partial operation $[\![\text{op}]\!]^{\text{pp}}_{\text{bnd}}$, operating only on *bounds*, and a corresponding *message-level* operation $[\![\text{op}]\!]_{\text{msg}}$. The operation $[\![\text{op}]\!]^{\text{pp}}_{\text{bnd}}$ is partial, so that it may throw an error of the estimated bound (e.g., ciphertext noise) is too large to carry out the operation. Figure 4 illustrates the operation on bounds. The message-level operation $[\![\text{op}]\!]_{\text{msg}}$ emulates

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid v \in [\![s]\!]\ (s \in \{\text{msg}, \text{plain}\}) \mid \text{op}(\overrightarrow{e_i}) \\ \text{Commands} & c & ::= & \text{skip} \mid x := e \mid c; c \mid \text{if } e \text{ then } c \text{ else } c \end{array}$$

Figure 5: ILA Syntax

the operator's intended semantics after interpretation. For example, the message-level operation for homomorphic multiplication is multiplication on messages, while the message-level operations for unary noise management operations on ciphertexts are the identity.

Validity of Models. A plain ILA model given by Definition 4.1 specifies the types of all operations, but does not encode whether the FHE scheme in question is *correct*; for example, whether homomorphic addition overflows given appropriate inputs. To encode correctness conditions for FHE, we require that the ILA model is additionally *valid*:

- Commutativity: for all secret parameters sp, every $\text{op} : \overrightarrow{s_i} \to s$ and arguments $\overrightarrow{v_i \in [\![s_i]\!]}$, we have that if $[\![\text{op}]\!]^{\text{topub(sp)}}_{\text{bnd}}(\overrightarrow{|v_i|^{\text{sp}}_{s_i}}) = b$, then $|[\![\text{op}]\!](\overrightarrow{v_i})|^{\text{sp}}_s \leq_s b$ and
$$\text{interp}^{\text{sp}}_s([\![op]\!](\overrightarrow{v_i})) = [\![\text{op}]\!]_{\text{msg}}(\overrightarrow{\text{interp}^{\text{sp}}_{s_i}(v_i)});$$
- Downwards Closed: If $\text{op} : \overrightarrow{s_i} \to s$, and if $[\![\text{op}]\!]^{\text{topub(sp)}}_{\text{bnd}}(\overrightarrow{b_i})$ is defined and equals b, then if $b_i' \leq_{s_i} b_i$ for all i, we have that $[\![\text{op}]\!]^{\text{topub(sp)}}_{\text{bnd}}(\overrightarrow{b_i'}) = b'$ with $b' \leq_s b$.

The first condition, commutativity, guarantees correctness by requiring that if the bounds chosen by operator op are defined and equals b, then the output of $[\![\text{op}]\!](\overrightarrow{v_i})$ is equally bounded by b, and the message-level result of $[\![\text{op}]\!](\overrightarrow{v_i})$ given by $\text{interp}^{\text{sp}}_s$ is equal to the value of $[\![\text{op}]\!]_{\text{msg}}$ evaluated on the message-level interpretations of its arguments. For homomorphic addition, this amounts to the fact that $\text{enc}(x) \oplus \text{enc}(y)$ is an encryption of $x + y$; while for bootstrapping, this condition encodes that $\text{bootstrap}(c)$ does not change the underlying value inside of c. The second condition, downwards-closedness, guarantees monotonicity of the bounds: if an operator states that an output bound is defined on its inputs, then a smaller output bound is defined on smaller outputs.

Throughout the rest of this section, fix an ambient valid ILA model $\mathcal{M}$. We additionally assume a nullary operator $\text{true} : () \to \text{msg}$, in order to evaluate control flow operators.

4.2 Syntax and Semantics

Now that we have a model, we now define the core language of ILA. The syntax is given in Figure 5. Expressions may either be variables, values of sort msg or plain, or applications of operators. We do not allow hardcoding ciphertexts into programs, as they must come from the context. Commands are standard for an imperative language. Note that the core language does not consider loops, while our surface language (e.g., in Figure 2) does; our concrete type checker only allows loops which can be statically unrolled with a finite bound.

We give big-step semantics to ILA programs in Figure 6. Given a substitution γ mapping variables to values in $[\![\text{msg}]\!] \cup [\![\text{plain}]\!] \cup [\![\text{cipher}]\!]$, the big-step rules for expressions relate substitutions and expressions to values. The rule for op checks that each input is of

$$\boxed{\langle \gamma, e\rangle \Downarrow v} \qquad \frac{\gamma(x) = v}{\langle \gamma, x\rangle \Downarrow v} \qquad \frac{}{\langle \gamma, v\rangle \Downarrow v}$$

$$\frac{\mathsf{op} : \overrightarrow{s_i} \to s \qquad (\forall i, \langle \gamma, e_i\rangle \Downarrow v_i \in [\![s_i]\!]) \qquad [\![\mathsf{op}]\!](\overrightarrow{v_i}) = v}{\langle \gamma, \mathsf{op}(\overrightarrow{e_i})\rangle \Downarrow v}$$

$$\boxed{\langle \gamma, c\rangle \Downarrow \gamma'} \qquad \frac{}{\langle \gamma, \mathsf{skip}\rangle \Downarrow \gamma} \qquad \frac{\langle \gamma, e\rangle \Downarrow v}{\langle \gamma, x := e\rangle \Downarrow \gamma[x \mapsto v]}$$

$$\frac{\langle \gamma, c_1\rangle \Downarrow \gamma_1 \qquad \langle \gamma_1, c_2\rangle \Downarrow \gamma_2}{\langle \gamma, c_1; c_2\rangle \Downarrow \gamma_2} \qquad \frac{\langle \gamma, e\rangle \Downarrow [\![\mathsf{true}]\!]() \qquad \langle \gamma, c_1\rangle \Downarrow \gamma'}{\langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle \Downarrow \gamma'}$$

$$\frac{\langle \gamma, e\rangle \Downarrow v \neq [\![\mathsf{true}]\!]() \qquad \langle \gamma, c_2\rangle \Downarrow \gamma'}{\langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle \Downarrow \gamma'}$$

Figure 6: Big-Step Semantics for ILA.

$$\begin{array}{lllll} \text{Bounds} & \alpha_s & \in & B_s & \\ \text{Types} & \tau & ::= & \mathsf{cipher}\ \alpha_{\mathsf{cipher}} & \mid\ \mathsf{plain}\ \alpha_{\mathsf{plain}} \\ & & \mid & \mathsf{msg}\ \alpha_{\mathsf{msg}} & \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau & \end{array}$$

$$\frac{\alpha \leq_s \alpha' \qquad s \in \{\mathsf{cipher}, \mathsf{plain}, \mathsf{msg}\}}{s\ \alpha \leq s\ \alpha'}$$

Figure 7: ILA Types and Subtyping

$$\boxed{\mathsf{pp}; \Gamma \vdash e : \tau} \qquad \frac{x : \tau \in \Gamma}{\mathsf{pp}; \Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\mathsf{pp}; \Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\mathsf{pp}; \Gamma \vdash e : \tau'}\ \textsc{sub}$$

$$\frac{\mathsf{sort}(v) = s}{\mathsf{pp}; \Gamma \vdash v : s\ |v|^{\mathsf{pp}}_s}\ \textsc{const}$$

$$\frac{\mathsf{op} : \overrightarrow{s_i} \to s \qquad (\forall i, \mathsf{pp}; \Gamma \vdash e_i : \tau_i \wedge \mathsf{sort}(\tau_i) = s_i) \qquad [\![\mathsf{op}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(|\overrightarrow{\tau_i}|) = \alpha}{\mathsf{pp}; \Gamma \vdash \mathsf{op}(\overrightarrow{e_i}) : s\ \alpha}\ \textsc{op}$$

Figure 8: ILA Typing: Expressions

the correct sort, and if so, evaluates $[\![\mathsf{op}]\!]$. The big-step rules for commands relate substitutions and commands to output substitutions, and are standard.

4.3 Type System

Types for ILA programs, and their associated rules for subtyping, are given in Figure 7. Each type τ is a pair of a sort s, and an associated bound $\alpha_s \in B_s$. Given a type τ, we write $|\tau|$ for the bound contained in the type. Subtyping in ILA requires that the two types are of the same sort and the first associated bound is less than the second, in the ordering induced by the sort.

Expressions are typed using Figure 8. The type checking judgment takes the public parameters pp as input to check that all bounds are correct. The rule for values computes the bound $|v|^{\mathsf{pp}}_s$ as part of the output type; since we only allow messages and encoded plaintexts in source programs, we do not need the secret parameters. For operators, we check that all inputs are sort-correct, and

$$\boxed{\mathsf{pp}; \Gamma \vdash c \dashv \Gamma'} \qquad \frac{\mathsf{pp}; \Gamma \vdash e : \tau}{\mathsf{pp}; \Gamma \vdash x := e \dashv \Gamma[x \mapsto \tau]}\ \textsc{Assgn}$$

$$\frac{}{\mathsf{pp}; \Gamma \vdash \mathsf{skip} \dashv \Gamma}\ \textsc{skip} \qquad \frac{\mathsf{pp}; \Gamma \vdash c_1 \dashv \Gamma' \qquad \mathsf{pp}; \Gamma' \vdash c_2 \dashv \Gamma''}{\mathsf{pp}; \Gamma \vdash c_1; c_2 \dashv \Gamma''}\ \textsc{seq}$$

$$\frac{\mathsf{pp}; \Gamma \vdash c_1 \dashv \Gamma_1 \qquad \mathsf{pp}; \Gamma \vdash c_2 \dashv \Gamma_2 \qquad \mathsf{pp}; \Gamma \vdash e : \mathsf{msg}\ \alpha \qquad \Gamma_1 \sqcap \Gamma_2 \sim \Gamma'}{\mathsf{pp}; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \dashv \Gamma'}\ \textsc{ite}$$

$$\boxed{\Gamma_1 \sqcap \Gamma_2 \sim \Gamma}$$

$$\frac{(\forall x \in \Gamma, x \in \Gamma_1 \wedge \Gamma_1(x) \leq \Gamma(x)) \qquad (\forall x \in \Gamma, x \in \Gamma_2 \wedge \Gamma_2(x) \leq \Gamma(x))}{\Gamma_1 \sqcap \Gamma_2 \sim \Gamma}$$

Figure 9: ILA Typing: Commands

then check that the output bound $[\![\mathsf{op}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(|\overrightarrow{\tau_i}|)$ is defined; if it is, we record this in the output type. In essence, since we have a valid ILA model (Section 4.1), the requirement that the output bound is correct enforces that all relevant quantities (e.g., RLWE noise) are within bounds.

Type checking for commands is given in Figure 9, and is largely standard. For if statements, we require that the branch is a cleartext message, and that both branches type check; then, we *merge* the two output contexts using the judgment $\Gamma_1 \sqcap \Gamma_2 \sim \Gamma$, which requires that Γ is a sound overapproximation of the post-states represented by Γ_1 and Γ_2.

4.4 Semantic Validity

Before proving full functional correctness, we prove *semantic validity*, which demonstrates that all runtime values are within bounds described by their types (as modeled by the ILA model). We do so via *semantic type soundness*, which requires we give a semantics to types:

$$[\![\mathsf{cipher}\ \alpha]\!]^{\mathsf{sp}} = \{c \in [\![\mathsf{cipher}]\!] \mid |c|^{\mathsf{sp}}_{\mathsf{cipher}} \leq_{\mathsf{cipher}} \alpha\}$$
$$[\![\mathsf{plain}\ \alpha]\!]^{\mathsf{sp}} = \{p \in [\![\mathsf{plain}]\!] \mid |p|^{\mathsf{sp}}_{\mathsf{plain}} \leq_{\mathsf{plain}} \alpha\}$$
$$[\![\mathsf{msg}\ \alpha]\!]^{\mathsf{sp}} = \{m \in [\![\mathsf{msg}]\!] \mid |m|^{\mathsf{sp}}_{\mathsf{msg}} \leq_{\mathsf{msg}} \alpha\}$$

Thus, $[\![\tau]\!]^{\mathsf{sp}}$ are the values of the correct sort that are soundly approximated by τ bounds. Note that while we use the secret parameters to define soundness, only the public parameters pp are required for type checking.

Now, we show that all runtime values of type τ must reside in $[\![\tau]\!]^{\mathsf{sp}}$. First, given a substitution γ, we say that $\mathsf{sp}; \Gamma \vDash \gamma$ if for all $x \in \Gamma$, $x \in \gamma \wedge \gamma(x) \in [\![\Gamma(x)]\!]^{\mathsf{sp}}$. Then, for expressions, say that $\mathsf{sp}; \Gamma \vDash e : \tau$ if for all $\mathsf{sp}; \Gamma \vDash \gamma$, $\langle \gamma, e\rangle \Downarrow v \wedge v \in [\![\tau]\!]^{\mathsf{sp}}$ for some v. Intuitively, $\mathsf{sp}; \Gamma \vDash \gamma$ states that γ is a well-formed substitution relative to its semantic types, while $\mathsf{sp}; \Gamma \vDash e : \tau$ states that, given a well-formed substitution, e evaluates to a well-formed value relative to $[\![\tau]\!]^{\mathsf{sp}}$. We now show semantic safety for expressions:

Theorem 4.1 (Semantic Safety: Expressions). *Suppose that* $\mathsf{topub}(\mathsf{sp}); \Gamma \vdash e : \tau$. *Then* $\mathsf{sp}; \Gamma \vDash e : \tau$.

$$\boxed{\langle \gamma, e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v} \qquad \frac{\gamma(x) = v}{\langle \gamma, x\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v} \qquad \frac{v \in [\![s]\!]}{\langle \gamma, v\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \mathsf{interp}^{\mathsf{sp}}_{s}(v)}$$

$$\frac{\mathsf{op} : \overrightarrow{s_i} \to s \qquad (\forall i, \langle \gamma, e_i\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v_i) \qquad [\![\mathsf{op}]\!]_{\mathsf{msg}}(\overrightarrow{v_i}) = v}{\langle \gamma, \mathsf{op}(\overrightarrow{e_i})\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v}$$

$$\boxed{\langle \gamma, c\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma'} \qquad \frac{}{\langle \gamma, \mathsf{skip}\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma} \qquad \frac{\langle \gamma, e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v}{\langle \gamma, x := e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma[x \mapsto v]}$$

$$\frac{\langle \gamma, c_1\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma_1 \qquad \langle \gamma_1, c_2\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma_2}{\langle \gamma, c_1; c_2\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma_2}$$

$$\frac{\langle \gamma, e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} [\![\mathsf{true}]\!]() \qquad \langle \gamma, c_1\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma'}{\langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma'}$$

$$\frac{\langle \gamma, e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v \neq [\![\mathsf{true}]\!]() \qquad \langle \gamma, c_1\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma'}{\langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} \gamma'}$$

Figure 10: Big-Step Semantics for Message Equivalence.

For commands, say that $\mathsf{sp}; \Gamma \vDash c \dashv \Gamma'$ if for all $\mathsf{sp}; \Gamma \vDash \gamma$, $\langle \gamma, c\rangle \Downarrow \gamma'$ and $\mathsf{sp}; \Gamma' \vDash \gamma'$. Intuitively, commands must map well-formed substitutions under Γ to well-formed substitutions under Γ':

Theorem 4.2 (Semantic Safety: Commands). *If* $\mathsf{topub}(\mathsf{sp}); \Gamma \vdash c \dashv \Gamma'$, *then we have that* $\mathsf{sp}; \Gamma \vDash c \dashv \Gamma'$.

4.5 Message Equivalence

Now that we have proven semantic safety, we prove functional correctness, which we formalize as *message equivalence*. Intuitively, message equivalence states that decrypting the result of any homomorphic circuit is equivalent to running that same circuit in cleartext, on decrypted inputs. This is the main guarantee of ILA.

To formalize message equivalence, we give an alternative big-step semantics to expressions and commands in Figure 10 that describes cleartext evaluation. Here, all substitutions map variables to cleartext values (i.e., values in $[\![\mathsf{msg}]\!]$). We then define cleartext evaluation by immediately interpreting (i.e., decoding) all encoded plaintexts into messages, and interpreting operators through their message-level semantics $[\![\mathsf{op}]\!]_{\mathsf{msg}}$. Hence, homomorphic addition will be interpreted as ordinary addition on integers, and bootstrapping operators will be interpreted as no-ops. The semantics for commands are similar to their ordinary big-step rules.

We now show that evaluating expressions and commands under their ordinary semantics, and then interpreting (e.g., decrypting), is equivalent to evaluating under the alternative semantics $\Downarrow^{\mathsf{sp}}_{\mathsf{msg}}$. To define the below theorems formally, we introduce an auxiliary definition: given a type context Γ and a substitution γ such that $\mathsf{sp}; \Gamma \vDash \gamma$, let

$$\mathsf{interp}^{\mathsf{sp}}_{\Gamma}(\gamma) = \{x \mapsto \mathsf{interp}^{\mathsf{sp}}_{\mathsf{sort}(\Gamma(x))}(\gamma(x))\}.$$

Thus, $\mathsf{interp}^{\mathsf{sp}}_{\Gamma}(\gamma)$ maps substitutions γ under the original FHE semantics into substitutions only over messages. With this definition, we can now prove message equivalence for expressions and commands:

		ILA	$\mathrm{ILA}_{(bgv)}$
			$\epsilon \in \mathbb{Q}^{+}_{\geq 1}$
			$\omega \in \mathbb{Z}^{+}$ $\quad \mathsf{inf}, \mathsf{sup} \in \mathbb{Q}$
Values	$v ::=$	$[\![\mathsf{msg}]\!]$	$n \in \mathbb{Z}^{d}_{t}$
		$\mid [\![\mathsf{plain}]\!]$	$\mid p \in \frac{\mathbb{Z}_t[x]}{(x^d+1)}$
Expressions	$e ::=$	$x \mid v$	$x \mid v$
		$\mid \mathsf{op}(\overrightarrow{e_i})$	$\mid e_1 \oplus e_2 \mid e_1 \otimes e_2$
			$\mid \mathsf{modswitch}(e)$
Statements	$c ::=$	$\mathsf{skip} \mid x := e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c$	
Types	$\tau ::=$	$\mathsf{cipher}\ \alpha_{\mathsf{cipher}}$	$\mathsf{cipher}\ \langle \mathsf{inf}, \mathsf{sup}, \epsilon, \omega\rangle$
		$\mid \mathsf{plain}\ \alpha_{\mathsf{plain}}$	$\mid \mathsf{plain}\ \langle \mathsf{inf}, \mathsf{sup}\rangle$
		$\mid \mathsf{msg}\ \alpha_{\mathsf{msg}}$	$\mid \mathbb{Z}^{d}_{t}$

Figure 11: $\mathrm{ILA}_{(bgv)}$ syntax instantiation. Columns ILA and $\mathrm{ILA}_{(bgv)}$ show corresponding syntax. Sorts $[\![s]\!]$, operator op and bounds α_s are instantiated accordingly. The row c denoting statements is common to both.

Theorem 4.3 (Message Equivalence: Expressions). *Suppose that* $\mathsf{topub}(\mathsf{sp}); \Gamma \vdash e : \tau$ *and* $\mathsf{sp}; \Gamma \vDash \gamma$. *Then,* $\langle \gamma, e\rangle \Downarrow v$ *and* $\langle \mathsf{interp}^{\mathsf{sp}}_{\Gamma}(\gamma), e\rangle \Downarrow^{\mathsf{sp}}_{\mathsf{msg}} v_{\mathsf{msg}}$ *such that* $\mathsf{interp}^{\mathsf{sp}}_{\mathsf{sort}(v)}(v) = v_{\mathsf{msg}}$.

The proof of Theorem 4.3 follows by induction on the typing derivation of e. The case for operators makes use of commutativity and downwards-closedness of the ILA model (Section 4.1) in order to interchange decryption and evaluation of the operator.

After proving message equivalence for expressions, we can then prove it for commands. The proof is a straightforward induction on the typing derivation for commands. Appendix C shows the proof sketch.

Theorem 4.4 (Message Equivalence: Commands). *Suppose that* $\mathsf{topub}(\mathsf{sp}); \Gamma \vdash c \dashv \Gamma'$ *and* $\mathsf{sp}; \Gamma \vDash \gamma$. *Then, if* $\langle \gamma, c\rangle \Downarrow \gamma_1$, *we have that* $\langle \mathsf{interp}^{\mathsf{sp}}_{\Gamma}(\gamma), c\rangle \Downarrow_{\mathsf{msg}} \gamma_1'$ *such that, for all* $x \in \gamma_1$, $\mathsf{interp}^{\mathsf{sp}}_{\mathsf{sort}(\gamma_1(x))}(\gamma_1(x)) = \gamma_1'(x)$.

5 ILA Models

At a high-level, the recipe for scheme instantiation involves defining all sets and mappings from Definition 4.1 and proving that the commutativity and downwards axioms hold.

5.1 $\mathrm{ILA}_{(bgv)}$: ILA for BGV Scheme

We introduce $\mathrm{ILA}_{(bgv)}$, an instantiation of ILA model for the BGV scheme, a levelled RLWE scheme [7]. Figure 11 shows the syntax correspondence between ILA (Figures 5 and 7) and $\mathrm{ILA}_{(bgv)}$. Abstract operators, sorts and bounds from ILA are instantiated. Operator op is instantiated with homomorphic multiplication $\otimes$, addition $\oplus$ and modulus switching modswitch. Plaintext and ciphertext bounds , α_{plain} and α_{cipher}, track the interval [inf, sup] in which the underlying message lies as well as the current noise ϵ. Intuitively, ϵ measures how far the coefficients are from corrupting the ciphertext. The level ω is drawn from a pre-defined set of levels $\{q_0, q_1, \ldots, q_L\}$. We say $((\mathsf{inf}, \mathsf{sup}), \epsilon, \omega) \leq_{\mathsf{cipher}} ((\mathsf{inf}', \mathsf{sup}'), \epsilon', \omega)$ if $\mathsf{inf}' \leq \mathsf{inf} \leq \mathsf{sup} \leq \mathsf{sup}'$ and $\epsilon \leq \epsilon'$.

Since noise ϵ is added during encryption, plaintext contains no noise. As a result plaintext-plaintext operations and plaintext ciphertext additions are not noise increasing operations. However,

noise increase after a plaintext-ciphertext multiplication is proportional to that of a fresh cipher-ciphertext multiplication.

Definition 5.1 summarizes the instantiation. Note that it closely follows ILA model defined in Section 4.

Definition 5.1 ($\text{ILA}_{(bgv)}$). *The $\text{ILA}_{(bgv)}$ model is defined as follows:*

- $\mathcal{SP} := \{t, d, \{q_\omega\}, pk, relin_k, \text{encode}, \text{decode}\} \times \{sk\}$
- $\mathcal{PP} = \{d, t, \{q_\omega\}, pk, relin_k, \text{encode}, \text{decode}\}$
- $\text{topub} = \mathcal{SP} = \mathcal{PP} \times \{sk\} \rightarrow \mathcal{PP}$ *is the projection onto* $\mathcal{PP}$
- *Sort* $\mathcal{S} = \{\text{msg}, \text{plain}, \text{cipher}\}$
  - $[\![\text{msg}]\!] = n \in \mathbb{Z}_t^d$
  - $[\![\text{plain}]\!] = p \in \frac{\mathbb{Z}_t[x]}{(x^d+1)}$
  - $[\![\text{cipher}]\!] = ct \in R_q^{n+1} \times \mathbb{Z}_{\geq 0}$
- *Encoding and decoding functions:*
  - encode: $\mathbb{Z}_t^d \rightarrow R_t$
  - decode: $R_t \rightarrow \mathbb{Z}_t^d$
- *Bounds sets:*
  - $B_{\text{msg}} = \emptyset$
  - $B_{\text{plain}} = \{(\text{inf}, \text{sup}) \mid \text{inf} \leq \text{sup}\}$ *ordered by* $\leq_{\text{plain}}$
  - $B_{\text{cipher}} = \{(\text{inf}, \text{sup}, \epsilon, \omega) \mid \text{inf} \leq \text{sup}$ *and* $\epsilon, \omega \in \mathbb{N}\}$ *ordered by* $\leq_{\text{cipher}}$
- *Mappings for bounds computation:*
  - $|m|^{\text{pp}}_{\text{msg}} = \emptyset$
  - $|p|^{\text{pp}}_{\text{plain}} = (\text{inf}, \text{sup})$ *(explained below)*
  - $|\text{ct}, \omega|^{\text{sp}}_{\text{cipher}} = (\text{inf}, \text{sup}, \epsilon, \omega)$ *(explained below)*
- *Decoding on plaintexts:* $\text{interp}^{\text{pp}}_{\text{plain}}(p) = \text{decode}(p)$
- *Decryption on ciphertexts:* $\text{interp}^{\text{sp}}_{\text{cipher}}(\text{ct}) = \text{decode}(\text{decrypt}(\text{ct}))$
- *Operator* $\text{op} \in \{\oplus, \otimes, \text{modswitch}\}$ *defined as follows:*
  - $[\![\otimes]\!]$: $\{\text{cipher}, \text{plain}\}, \{\text{cipher}, \text{plain}\} \rightarrow \{\text{cipher}, \text{plain}\}$
  - $[\![\oplus]\!]$: $\{\text{cipher}, \text{plain}\}, \{\text{cipher}, \text{plain}\} \rightarrow \{\text{cipher}, \text{plain}\}$
  - $[\![\text{modswitch}]\!]$: $\text{cipher} \rightarrow \text{cipher}$
- *Operator bounds manipulation mapping,* $[\![\text{op}]\!]^{\text{pp}}_{\text{bnd}}$ *defined in Table 1.*

The set $\mathcal{SP}$ contains all scheme parameters including plaintext modulus t, poly modulus degree d, scheme-related keys as well as encoding and decoding functions. The public parameter set $\mathcal{PP} \subseteq \mathcal{SP}$ contains all public parameters including public and relinearization keys. The function topub projects $\mathcal{SP}$ to $\mathcal{PP}$.

Bounds sets for plain and ciphertexts include set of elements, α_{plain} and α_{cipher} (described above). Bounds set for messages is empty. In bounds computations for plaintext p if $z = \text{decode}(p)$ then inf and sup are two random rationals such that $\text{inf} \leq \min_i\{z_i\} \leq \max_i\{z_i\} \leq \text{sup}$. Bound computations for ciphertext involves decrypting the ciphertext using the secret key to obtain minimum and maximum of the underlying cleartext. If eval_noise_{sk} computes noise in a ciphertext then ϵ is chosen (deterministically or randomly) such that $\text{eval\_noise}_{sk}(\text{ct}) \leq \epsilon$. We also assume (but not explicitly define) the decoding and decryption functions. Though these are BGV scheme primitives, they are outside $\text{ILA}_{(bgv)}$ model as these operations are typically used at the end of the circuit evaluation.

Table 1 provides the bounds computations for the key $\text{ILA}_{(bgv)}$ homomorphic operations. The second column computes the operator bound $[\![.]\!]^{\text{pp}}_{\text{bnd}}$ and the last column describes the conditions under which the bound is defined. For example, $\text{ct}_1 \oplus \text{ct}_2$ with input bounds $(\text{inf}_i, \text{sup}_i, \epsilon_i, \omega)$ results in the bound $(\text{inf}_1 + \text{inf}_2, \text{sup}_1 + \text{sup}_2, \epsilon_1 + \epsilon_2, \omega)$ provided $-t/2 \leq \text{inf} \leq \text{sup} < t/2$ and $\epsilon \leq \frac{q_\omega}{2}$ where t and q_ω are the plaintext modulus and critical value at modulus level ω, respectively. Operator modswitch decrements the modulus level by one and reduces the original noise by $\frac{q_{\omega-1}}{q_\omega}$; a small correction B_r is also added to correct the rounding error induced by the division.

Our noise estimation for homomorphic multiplication is parametric in functions f and g with the only restriction that they are downwards closed. Thus, they are positive and monotonic. In Section 7, we refer to estimators from the existing literature that satisfy these conditions.

5.1.1 $\text{ILA}_{(bgv)}$ Typing Rules.

$\text{ILA}_{(bgv)}$ typing rules are obtained by instantiating ILA type system from Figure 8 using the bounds information computed in Table 1. Below, we show a sample typing rule for $\otimes$ operator that multiplies two ciphertexts.

$$\frac{\begin{array}{c}\otimes : \text{cipher}, \text{cipher} \rightarrow \text{cipher} \\ \forall i, \text{pp}; \Gamma \vdash e_i : \text{cipher}\ \langle \text{inf}_i, \text{sup}_i, \epsilon_i, \omega \rangle \\ -t/2 \leq \text{inf} \leq \text{sup} < t/2 \qquad f(\epsilon_1, \epsilon_2) \leq q_\omega/2\end{array}}{\text{pp}; \Gamma \vdash e_1 \otimes e_2 : \text{cipher}\ \langle \text{inf}, \text{sup}, f(\epsilon_1, \epsilon_2), \omega \rangle}$$

The output interval $[\text{inf}, \text{sup}]$ is computed from Table 1. The exact definition of f is dependent on the noise estimation technique. This enables an extensible framework for plug-and-play noise estimators. Appendix E shows typing rule instantiation for modswitch.

5.1.2 Instantiation of ILA Functional Correctness.

To complete the $\text{ILA}_{(bgv)}$ instantiation, we have to show that commutativity and downwards closure axioms hold in our instantiation. Downwards closed axiom is immediate as the noise growth in homomorphic operations is monotonic.

The commutativity axiom partly encodes the correctness theorems of the BGV scheme. Intuitively, it says that the cipher addition of two ω level ciphers with noise ϵ_1 and ϵ_2 is a valid ciphertext i.e., decryption yields a *correct* plaintext value when $\epsilon_1 + \epsilon_2 \leq q_\omega/2$. Similarly, the cipher multiplication of such ciphertexts is valid when $\epsilon_1 * \epsilon_2 \leq q_\omega/2$ (cf. Lemma 7 and 8 from Section 5 [7]). Further, $-t/2 \leq \text{inf}_1 + \text{inf}_2 \leq \text{sup}_1 + \text{sup}_2 < t/2$ guarantees that the resultant plaintext decodes correctly. Correctness up to plaintext space follows Lemma 7 and 8 from Section 5 [7]. The rest is immediate by induction of typing derivation.

Since $\text{ILA}_{(bgv)}$ is a valid ILA model, we get Theorem 4.4 for free. That is if the initial environment contains correct-by-construction ciphertexts, then well-typed $\text{ILA}_{(bgv)}$ programs are functionally correct (up to the correctness of FHE primitive operations). Specifically, the programs are devoid of noise overflow errors. Additionally, the decryption and decode of the output(s) exactly matches the corresponding cleartext values, i.e., the absence of any plaintext modulus wraparound errors. We thus guarantee the functional correctness of a BGV circuit via type checking.

Note the assumption on the initial environment. If the ciphertexts are adversarial, i.e., they do not match the input types, then the theorem guarantees are invalid. Thus, the input sources are trusted. This is a reasonable assumption as FHE circuits run on sensitive but trusted inputs.

Op	$[\![\cdot]\!]^{\mathsf{pp}}_{\mathsf{bnd}}$		Conditions
$\oplus$	$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\inf_i, \sup_i, \epsilon_i, \omega})$	$= (\inf, \sup, \epsilon, \omega)$	if $-t/2 \le \inf \le \sup < t/2$ and $\epsilon \le \kappa$
	$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\inf_i, \sup_i})$	$= (\inf, \sup)$	if $-t/2 \le \inf \le \sup < t/2$
	$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}((\inf_1, \sup_1), (\inf_2, \sup_2, \epsilon_2, \omega))$	$= (\inf, \sup, \epsilon, \omega)$	if $-t/2 \le \inf \le \sup < t/2$ and $\epsilon \le \kappa$ where $\inf := \inf_1 + \inf_2$, $\sup := \sup_1 + \sup_2$, $\epsilon := \epsilon_1 + \epsilon_2$
$\otimes$	$[\![\otimes]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\inf_i, \sup_i, \epsilon_i, \omega})$	$= (\inf, \sup, f(\epsilon_1, \epsilon_2), \omega)$	if $-t/2 \le \inf \le \sup < t/2$ and $f(\epsilon_1, \epsilon_2) \le \kappa$
	$[\![\otimes]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\inf_i, \sup_i})$	$= (\inf, \sup)$	if $-t/2 \le \inf \le \sup < t/2$
	$[\![\otimes]\!]^{\mathsf{pp}}_{\mathsf{bnd}}((\inf_1, \sup_1), (\inf_2, \sup_2, \epsilon, \omega))$	$= (\inf, \sup, g(\epsilon), \omega)$	if $-t/2 \le \inf \le \sup < t/2$ and $g(\epsilon) \le \kappa$ where $\inf := \min\{\inf_1 * \inf_2, \inf_1 * \sup_2, \sup_1 * \inf_2, \sup_1 * \sup_2\}$, $\sup := \max\{\inf_1 * \inf_2, \inf_1 * \sup_2, \sup_1 * \inf_2, \sup_1 * \sup_2\}$
modswitch	$[\![\mathsf{modswitch}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\inf, \sup, \epsilon, \omega)$	$= (\inf, \sup, \epsilon', \omega - 1)$	$0 \le \omega - 1 \le L - 1$; $\epsilon' = \frac{q_{\omega-1}}{q_\omega}\epsilon + B_r \le l_{\omega-1}$

Table 1: $\mathrm{ILA}_{(bgv)}$: ILA model for BGV and BFV scheme illustrating key homomorphic operations. Constant κ is $\frac{q_\omega}{2}$ for BGV, and $\frac{1}{2}$ for BFV; scheme parameters t and q_ω are plaintext modulus and coefficient modulus at modulus level ω, respectively.

5.2 $\mathrm{ILA}_{(bfv)}$: ILA for BFV Scheme

BFV is a levelled RLWE scheme that is highly similar to BGV except that messages are scaled up and encoded from the most significant bit. $\mathrm{ILA}_{(bfv)}$ is the instantiation of ILA for BFV scheme. For brevity, we only focus on key differences.

In a BFV scheme modulus switching is used as a tool to improve computational performance. Since this is orthogonal to the current focus, $\mathrm{ILA}_{(bfv)}$ supports all operations of $\mathrm{ILA}_{(bgv)}$ except modswitch. Thus, the instantiation is similar to that of $\mathrm{ILA}_{(bgv)}$ with two differences. First, the cipher bounds set B_{cipher} is the set of triples, $(\inf, \sup, \epsilon)$ with the standard lexicographic ordering. Note that modulus level is not included. Second, the operator bounds manipulation mapping uses the condition $\epsilon \le \frac{1}{2}$ instead of $\epsilon \le \frac{q_\omega}{2}$ (the last column of Table 1).

It is obvious that $\mathrm{ILA}_{(bfv)}$ also satisfies commutativity and downwards closed axioms, and thus we get correctness, that is, Theorem 4.4, for free.

5.3 $\mathrm{ILA}_{(tfhe)}$: ILA for TFHE Scheme

We introduce $\mathrm{ILA}_{(tfhe)}$, an instantiation of ILA model for the CGGI/TFHE scheme [11]. TFHE is a fast bootstrapping based scheme. It differs from BGV in that it supports additional types of ciphertexts (LWE, RLWE and RGSW) and the corresponding operations including boolean (e.g., exclusive-or). Plaintext and ciphertexts are elements of the sets R_t and $\mathbb{Z}_q^{n+1} \cup R_q^{n+1}$, respectively (see Section 2). LWE and RLWE ciphertexts are drawn from the sets $\mathbb{Z}_q^{n+1}$ and R_q^{n+1}, respectively; a GLWE ciphertext is either a LWE cipher or a RLWE cipher. A RGSW ciphertexts can be seen as elements of the space $R_q^{n+1 \times n+1}$.

RGSW ciphertexts should be considered as tools necessary to support various internal TFHE operations that are not exposed to the user. Hence, a ciphertext is an element of either $\mathbb{Z}_q^{n+1}$ or R_q^{n+1}, and we track the set using *id*. Thus, we define α_{cipher} as $(id, \inf, \sup, \epsilon)$. We say $(id, \inf, \sup, \epsilon) \le_{\mathsf{cipher}} (id', \inf', \sup', \epsilon')$ if $id = id'$ followed by the standard lexicographic ordering of the remaining triples. The bounds set for plaintext is the same as that of BGV. Similarly, a plaintext could be an element of $\mathbb{Z}_t$ or R_t, however the former coincides with the cleartext space. As a result we assume a plaintext is always a RLWE plaintext.

TFHE supports programmable bootstrapping (PBS), pbs, for noise reduction in the ciphertext. Recall from Section 2 that programmable bootstrapping is a composition of modulus switching, blind rotation and sample extraction operations. For PBS to succeed, the switching operation must succeed; the latter requires a non-zero noise budget.

TFHE also supports internal product, $\boxtimes$ that operates exclusively on GLWE ciphertexts and an external product, $\boxdot$, that operates on (restricted) combinations of ciphertexts. TFHE supports more operators, however, these additional operators are implemented as a combination of basic operators. For example, controlled multiplexer, cmux, that conditionally selects a ciphertext is implemented as a combination of external product, GLWE cipher addition and multiplication. Definition 5.2 summarizes the key details and focuses on differences with respect to the BGV scheme. We provide the full instantiation in Appendix D.

Definition 5.2 ($\mathrm{ILA}_{(tfhe)}$). *The $\mathrm{ILA}_{(tfhe)}$ model is defined as follows:*

- *Sort* $\mathcal{S} = \{\mathsf{msg}, \mathsf{plain}, \mathsf{cipher}\}$
  - $[\![\mathsf{msg}]\!] = n \in \mathbb{Z}_t$
  - $[\![\mathsf{plain}]\!] = p \in \frac{\mathbb{Z}_t[x]}{(x^d+1)}$
  - $[\![\mathsf{cipher}]\!] = ct \in R_q^{n+1} \cup \mathbb{Z}_t^{n+1}$
- *Bounds sets:*
  - $B_{\mathsf{cipher}} = \{(\mathsf{id}, \inf, \sup, \epsilon) \mid \inf \le \sup,$ $\mathsf{id} \in \{\mathsf{LWE}, \mathsf{RLWE}, \mathsf{RGSW}\}$ *and* $\epsilon \in \mathbb{N}\}$
- *Mapping for bounds computation:*
  - $|\mathsf{ct}|^{\mathsf{sp}}_{\mathsf{cipher}} = (\mathsf{id}, \inf, \sup, \epsilon)$ *(explained below)*
- *Operator* $\mathsf{op} \in \{\oplus, \otimes, \boxtimes, \boxdot, \mathsf{pbs}\}$. *Select operators defined as follows:*
  - $[\![\boxdot]\!] : \{\mathsf{cipher}, \mathsf{cipher}\} \to \{\mathsf{cipher}\}$
  - $[\![\mathsf{pbs}]\!] : \{\mathsf{cipher}, \mathsf{cipher}\} \to \{\mathsf{cipher}\}$
- *Operator bounds manipulation mapping,* $[\![\mathsf{op}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}$ *defined in Table 2.*

Similar to $\mathrm{ILA}_{(bgv)}$, $\mathrm{ILA}_{(tfhe)}$ defines scheme parameters as well as bounds sets. For a cipher *ct* the map $|.|^{\mathsf{sp}}_{\mathsf{cipher}}$ is defined as a tuple $(\mathsf{id}, \inf, \sup, \epsilon)$, where $\mathsf{id} = \mathsf{LWE}$ if the coefficients of the given cipher are elements of $\mathbb{Z}_t$ and $\mathsf{id} = \mathsf{RLWE}$ if the coefficients are elements of R_q^{n+1}. Value bounds inf and sup are random rational

numbers such that inf $\leq$ decrypt$(ct) \leq$ sup. The noise in ct is bounded by ϵ i.e., eval_noise$_{sk}(ct) \leq \epsilon$. Table 2 defines the operator bounds manipulation mappings for select TFHE operations. Despite the differences in the types of ciphertexts, the bounds manipulation is mostly straightforward for all types of ciphertext multiplication. The output noise is parameterized by the functions that are downward closed; the operation is defined iff the output noise less than $\frac{q}{t}$.

Given two ciphertexts ct_1 and ct_2, the pbs operator applies the function (e.g., LUT) in ct_1 to the second input ct_2. At the same time it resets the noise in the output to the nominal level (constant noise denoted by ϵ_b). The operation emits a LWE ciphertext with new bounds provided there are no overflows. The condition $-t/2 < \text{inf}_0 < \text{sup}_0$ ensures that there is no overflow in the output value. Note that it is sufficient to check that inf_0 and sup_0 as they are the value bounds of the LUT in ct_1. We can ignore inf and sup from the input ct_2. Interestingly, the constraint on the input noise $\epsilon < q/t - \delta$ is required to ensure that the switching step of the bootstrapping operation succeeds (see Section 2); δ is a constant representing switching error, thus ϵ must have a non-zero noise budget for PBS to succeed.

The commutativity and downwards closure axioms are similar to that of ILA$_{(bgv)}$. Since ILA$_{(tfhe)}$ is a valid ILA model, we get Theorem 4.4 for free.

6 Transformation Validation

ILA's type system can be used to validate that the transformations are bounds-preserving. For example, transforming $x_1 \otimes x_2 \otimes x_3 \otimes x_4$ to $y_1 \otimes y_2$ where $y_1 = x_1 \otimes x_2$ and $y_2 = x_3 \otimes x_4$ is not bounds-preserving: the former has a multiplicative depth of 3 where as the latter has the multiplicative depth of 2; this leads to different noise levels in the output. This is useful in building correct-by-construction optimizations.

We describe an optimization that infers the placement of modulus switching operations for loop-free programs and uses the type system to validate the bounds-preservation property.

6.1 Modulus Switch Inference

Recall from Section 2 that modulus switching refreshes noise. Though it reduces both noise and noise budget in a ciphertext, the relative reduction in noise is greater than that of noise budget. This is significant and somewhat surprising: if a program turns on switching level for every ciphertext, then the noise budget gets quickly exhausted reducing the circuit's overall multiplicative depth (compared to the vanilla circuit). Moreover, the operation is computationally expensive.

However, modulus switching could improve the circuit's multiplicative depth. Our inference algorithm identifies the conditions under which modulus switch placement succeeds. Toward this end, the algorithm iteratively estimates the placement and uses the type checker to validate the placement at every iteration.

Consider the program shown on the left of Figure 12. It computes $c1^{16}$. Suppose that c5 in line 4 yields incorrect output due to the noise overflow. Transforming line 4 to c5 = modswitch(c4 $\otimes$ c4) will not help as switching cannot recover any additional noise budget after-the-fact. So where should the placement be such that the computation succeeds? Our novel insight is to place the switching

Algorithm 1: Modulus Switch Inference

```
   // Create a definition list.
1  defList ← getDefList(ast);
   // Create a MD Tree for a given node.
2  mdtree ← MulDepthTree(node, φ);
3  ast ← Modswitch(ast, mdtree, Γ);
4  Function Modswitch (ast, mdtree, Γ) → AST:
       // Pick a leaf node.
5      node ← Leaves(mdtree);
6      p ← Parent(node, mdtree);
7      rhs ← defList[p];
8      ast, rhs' ← MSinsert(ast, p, rhs);
9      _, τ ← TypeCheck(rhs', Γ);
10     Γ ← Γ[node ↦ τ];
11     node ← p;
12     while node ≠ root(ast) do
13         p ← Parent(node, mdtree);
14         rhs ← defList[p];
15         ast, rhs ← MSLevel(ast, p, rhs, Γ);
16         stat, τ ← TypeCheck(rhs, Γ);
17         if !stat then
18             return fail;
19         Γ ← Γ[node ↦ τ];
20         node ← p;
21     return ast;
   // Type check an expression.
22 Function TypeCheck (e, Γ) → (bool, τ):
23     status ← ...;
24     return status, τ;
   // Ensure that operands have same modulus level.
25 Function MSLevel (ast, p, rhs, Γ) → (bool, τ):
26     rhs ← ...;
27     ast ← ...;
28     return ast, rhs;
29 Function MulDepthTree(node, wl):
30     rhs ← defList[node];
31     wl ← wl ∪ vars(rhs);
32     foreach xnode ∈ wl do
33         if xnode ∈ dom(defList) then
34             xtree ← MulDepthTree(xnode, φ);
               // Update children set for node
35             node.child ← node.child ∪ xtree;
       // returns MD Tree rooted at node
36     return node;
```

operations at *least positive multiplicative depth*. This yields maximal noise budget savings that could enable more operations. In this case, the multiplicative depth of c5 is 15 (since c5 = $c1^{16}$ with 15 multiplications) and that of c2 is 1. Thus, switching c2 yields maximal noise budget gains.

Op	$[\![\cdot]\!]^{\mathsf{PP}}_{\mathsf{bnd}}$	Conditions
$[\![\otimes]\!]^{\mathsf{PP}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{LWE}, \inf_i, \sup_i, \epsilon_i})$	$= (\mathsf{LWE}, \inf, \epsilon_b)$	ϵ_b is nominal $(= 1)$
$[\![\boxtimes]\!]^{\mathsf{PP}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{RGSW}, \inf_i, \sup_i, \epsilon_i})$	$= (\mathsf{RGSW}, \inf, \sup, f(\epsilon_1, \epsilon_2))$	$f(\epsilon_1, \epsilon_2) \leq q/t$
$[\![\boxdot]\!]^{\mathsf{PP}}_{\mathsf{bnd}}(\mathsf{RGSW}, \inf_1, \sup_1, \epsilon_1), (\mathsf{id}, \inf_2, \sup_2, \epsilon_2)$	$= (\mathsf{RLWE}, \inf, \sup, g(\epsilon_1, \epsilon_2))$	$g(\epsilon_1, \epsilon_2) \leq q/t$
$[\![\mathsf{pbs}]\!]^{\mathsf{PP}}_{\mathsf{bnd}}((\mathsf{RGSW}, \inf_0, \sup_0, \epsilon_0), (\mathsf{LWE}, \inf, \sup, \epsilon))$	$= (\mathsf{LWE}, \inf, \sup, \epsilon_b)$	$\max\{\log_2 \mid \inf \mid, \log_2 \mid \sup \mid\} \leq (\log_2(t) - 1)$ $\epsilon < q/t - \delta$ and $-t/2 \leq \inf_0 \leq \sup_0 < t/2$

Table 2: ILA$_{(tfhe)}$: ILA model for TFHE scheme illustrating select TFHE operations. Input ciphertext operand has the type $(id_i, \inf_i, \sup_i, \epsilon_i)$

```
1 c2 = c1 ⊗ c1                1 c2 = c1 ⊗ c1
2 c3 = c2 ⊗ c2                2 c3 = modswitch(c2) ⊗ modswitch(c2)
3 c4 = c3 ⊗ c3                3 c4 = c3 ⊗ c3
4 c5 = c4 ⊗ c4                4 c5 = c4 ⊗ c4
```

Figure 12: Modulus switch inference on the left program yields the right program.

However, there are two alternatives. First, changing line 1 to c2 = modswitch(c1 ⊗ c1); this causes the modulus level of c2 to be one less than that of c1. Second, changing line 2 to c3 = modswitch(c2) ⊗ modswitch(c2); this leaves the level of c2 unchanged.

Recall that switched operands can only operate with operands at the same level (Section 2). Thus, the former approach initiates a modulus level propagation chain: every use of c2 must ensure that the remaining operands are also at the same modulus level; otherwise the operation fails. However, the latter approach leaves the level of c2 unchanged; this allows program to use c2 as much as possible minimizing the number of changes.

Algorithm 1 presents the modulus switch inference algorithm. The input abstract syntax tree (AST) is assumed to be in Static Single Assignment (SSA) form, a reasonable assumption, as many widely used compiler intermediate representations (IRs) adopt SSA to facilitate optimization.

At a high level, the algorithm constructs a Multiplicative Depth Tree (MD Tree) for a cipher variable and inserts switching operations along the path with the highest multiplicative depth. The function `MulDepthTree` constructs the MD tree. The multiplicative depth is defined as the maximum distance from the root to any leaf node. Given a node `node`, the tree is built by recursively identifying the multiplicative operands that contribute to `node` and representing them as its child nodes. Additive operands are ignored, as they do not directly contribute to multiplicative depth; instead, their corresponding MD subtrees are reused in the construction. For example, the MD tree for c_7 in the program $c_1 := c_2 \otimes c_3; c_4 := c_5 \otimes c_6; c_7 := c_4 \oplus c_1$, c_4 is as follows: c_7 is the root node with c_2, c_3, c_5 and c_6 as the children. Note that c_4 and c_1 aren't used; instead their MD trees are used.

The inference function, `Modswitch`, picks a leaf node in the MD tree; inserts modulus switch operations on the operands of the node's parent; and updates the type of the node to reflect the correct modulus level. Modulus levelizer, `MSLevel`, transforms all the reaching definitions of the node to use the same modulus level. The definition of `MSLevel` and `TypeCheck` are straightforward and are thus omitted. Note that every AST transformation uses ILA 's type system as a validation framework.

The worst-case runtime complexity of the algorithm is $O(\ell \cdot n)$, where ℓ is the length of the modulus chain and n is the number of multiplications in the circuit. In other words, each multiplication may undergo up to ℓ modulus switch operations. Although achieving optimality is not a goal of this work, the algorithm aims to minimize the number of switching operations while preserving the correctness of the transformation.

Our inference algorithm is designed for loop-free programs; therefore, any loops must be unrolled prior to the inference step. For example, suppose the last iteration of the PSI program (Figure 2) causes an overflow. In this case, the inference process would insert modulus switching operations into lines 14–16 of the unrolled version of the program. In practice, nested loop depth is typically less than 5; as such, loops can be unrolled without significantly increasing runtime complexity.

7 Implementation

We extend the core language in Figure 5 with scheme-specific homomorphic operations as well as with other user-friendly features such as finite loops, vectors and matrices in a straightforward manner. Note that all loops have to terminate; otherwise the program violates Theorem 4.4. Under the hood, matrices are implemented as vectors. Interestingly, indexing vector might lead to out-of-bounds accesses if the index is not a constant. Since all ILA programs are terminating, a simple static analysis can help detect OOB accesses. However, it is orthogonal to the current work.

We implement ILA as a retargetable compiler with support for three popular backends—SEAL, OpenFHE and TFHE-rs—for executing the FHE circuits. It supports BGV, BFV and TFHE schemes. The compiler is implemented in Python and invokes backends either by using Python bindings (OpenFHE and SEAL) or by generating Rust code (TFHE-rs). During the setup phase, it instantiates all parameters and constructs the initial environment; statically type checks the environment and program; and if successful, invokes the corresponding backend to execute the circuit.

There are a couple of implementation challenges. First, FHE circuits inherently involve arbitrarily large number arithmetic (e.g., the coefficient modulus can be as large as 10^{250}). Fortunately, Python supports accurate arithmetic for integers. For floating point calculations, we rely on high-precision libraries such as numpy. Second, to determine the noise overflow statically, we need a heuristic estimate to estimate noise growth. In our current implementation

we employed the worst case—most conservative—noise heuristics for ILA$_{(bfv)}$ [17], ILA$_{(bgv)}$ [12] and average case heuristics for TFHE [11].

8 Evaluation

Our evaluation aims to answer two broad questions. First, *is ILA expressive*, in the sense of handling complex FHE circuits? Second, *is ILA efficient?*

8.1 Expressiveness

We evaluated a series of benchmarks that varied in both multiplicative depth (up to approximately 22 levels) and circuit breadth (up to approximately 8000 multiplicative gates). The breadth of a circuit refers to the number of multiplicative gates it contains. While ILA is theoretically capable of supporting circuits with arbitrary depth and breadth, in practice, its capabilities are limited by the constraints of the underlying cryptographic libraries.

Achieving higher multiplicative depth requires a larger coefficient modulus. In Microsoft SEAL, the maximum supported bit count for the coefficient modulus is 881 bits [3], which, in our experiments, corresponded to a maximum multiplicative depth of 21. OpenFHE, employs the Residue Number System (RNS) to represent the coefficient modulus. Although the use of the Residue Number System (RNS) enables support for larger coefficient moduli, it also incurs significantly higher memory overhead [21] and necessitates larger cache sizes for efficient execution [5]. In our experiments, OpenFHE programs with a multiplicative depth greater than 22 could not be executed successfully.

The breadth of a circuit is also limited by ciphertext size. Each ciphertext is represented as a degree-d polynomial, making its size proportional to d. For instance, when $d = 32768$, storing 8000 distinct ciphertexts would require more than 50 GB of memory.

Our evaluation includes all combinations of cipher-cipher and cipher-plain operations. The benchmarks also include computing functions for exponentiation (proxy for multiplicative depth) and fibonacci (proxy for additive depth). This shows that ILA compiler can handle large FHE circuits that are both wide and deep.

Additionally, to demonstrate that ILA can handle complex FHE circuits, we implemented a *private set intersection* (PSI), *private information retrieval* (PIR) and an image processing application. Each of these applications have complex control flow operations involving nested (finite) loops and use rich data structures such as vectors and matrices. PIR models database with 1000 rows.[2] All entries of the database are packed into a plaintext vector and the query as a ciphertext vector containing the secret index. A cipher-plain vector multiplication yields the required information. The image processing application uses matrix (size 1000x1000) operations to apply a filter (can be encrypted or plaintext) to the encrypted image. Note that for larger circuits, we optionally rely on an encoding that packs multiple ciphertexts reducing the size of the circuit.

In all these cases, ILA is able to detect overflows when the multiplicative depth exceeds the noise budget. When type checking succeeds, the compiler invokes the corresponding backend to execute the circuit. Note that all applications can run on multiple backends using different schemes. For example, PIR can run on both SEAL and OpenFHE by selecting BGV/BFV scheme; PSI can run on all backends for all schemes.

[2]Similar to *Wide Database* of Apple's caller ID PIR [1].

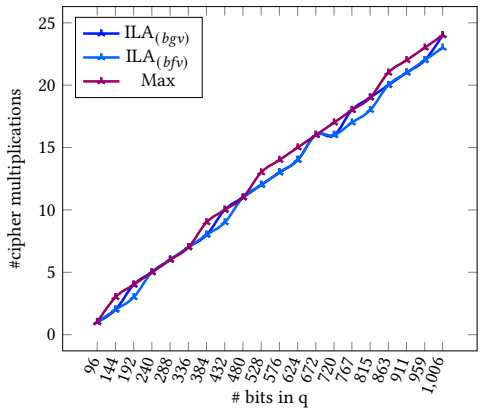

Figure 13: Multiplicative depth of ILA vs possible maximum.

8.2 Efficiency

A type checker that rejects all programs is trivially sound but not useful. In other words, too many false positives limit the practical use. To demonstrate the practicality of ILA, we evaluated ILA's instantiations against various measures.

Since noise estimators employed by ILA type checker are conservative, we evaluated if the type checker admits practical FHE circuits. We performed various experiments to measure how tight ILA's noise estimators are for ILA$_{(bgv)}$ and ILA$_{(bfv)}$. This is measured using the multiplicative depth—the maximal number of sequential homomorphic multiplications that can be performed on fresh ciphertexts without noise overflow. To a good approximation, it can be used as a proxy for the lower bound of the circuit's size [6, 25].

For TFHE, since noise is less of a concern, we measured the runtime of value overflow detection. Figures 13 and 16 summarize the results.

Multiplicative depth. To compute the multiplicative depth for a given coefficient modulus q, we multiply a sequence of fresh ciphers until ILA type checker detected a noise overflow. Using the same scheme parameters, we check whether is a true noise overflow in both SEAL and OpenFHE libraries. Note that the libraries may admit higher multiplicative depth and we record the maximal depth. Figure 13 plots the multiplicative depth obtained by varying the coefficient modulus q. It shows that ILA's multiplicative depth closely matches the actual maximal multiplicative depth for both BGV and BFV schemes.

Given that OpenFHE is a industrial strength library, we refine the above evaluation by comparing the multiplicative depth of ILA and OpenFHE. Figure 14 shows that ILA always outperforms OpenFHE. This demonstrates that ILA's noise overflow analysis is useful and admits practical FHE circuits.

Multiplicative depth of OpenFHE vs ILA. Recall that OpenFHE guarantees correctness of an FHE circuit up to a preselected multiplicative depth. There are two main problems with their estimates, their estimates are not tight, and they treat cipher-cipher and plain-cipher multiplications as the same. Note that plaintext does not

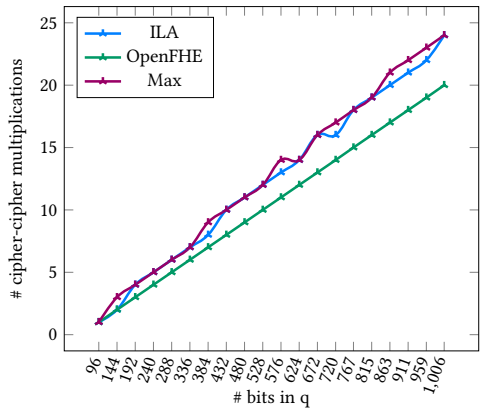

Figure 14: Multiplicative depth of ILA$_{(bgv)}$ vs OpenFHE.

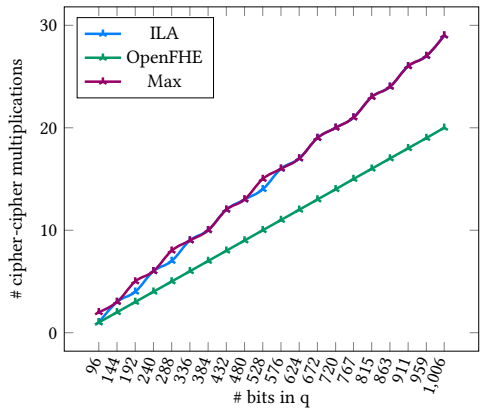

Figure 15: Plain-cipher mult depth of ILA$_{(bgv)}$ vs OpenFHE

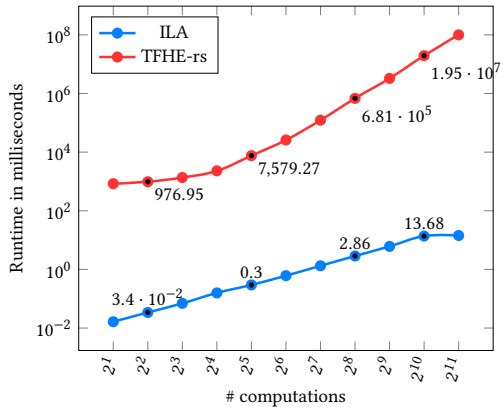

Figure 16: ILA$_{(tfhe)}$ vs TFHE value overflow runtimes.

carry any noise. As a result plain-cipher multiplication result has less noise than a cipher-cipher multiplication. Hence, more plain-cipher multiplications are possible than OpenFHE guarantees. In this experiment, we computed plain-cipher multiplications for BGV. Figure 15 shows that ILA over performed OpenFHE for every value of q. Appendix B shows full evaluation of ILA for BGV and BFV schemes.

Value overflow. In the case of TFHE, the multiplication operation resets the noise in the result. Moreover, the noise growth is very slow with other homomorphic operations. Hence, runtime performance of value overflow detection is a greater concern.

Our goal is to compare the runtime performance of value overflow detections in TFHE and ILA$_{(tfhe)}$. To this end, we construct a FHE circuit with $t = 2^p$ additions. Note that this allows $t - 1$ valid additions and overflow occurs with the last addition. The circuit is implemented in both ILA and TFHE-rs. For the latter, we use *overflow_add* that tracks the value overflows during the circuit execution. We measure the time taken to detect overflows for the same number of operations in both ILA ad TFHE-rs.

Figure 16 shows the runtime comparison of ILA$_{(tfhe)}$ and TFHE-rs for detecting value overflows when run on a Intel i9 2.3 GHz 8-core processor with 16 GB RAM. Note that we use the average (1000 runs) execution time for ILA$_{(tfhe)}$ type checker to offset any randomness in the runs.

Evaluation shows that TFHE-rs is orders of magnitude slower than ILA$_{(tfhe)}$. In all instances, ILA$_{(tfhe)}$ was able to detect value overflows in milliseconds, whereas TFHE-rs took hours for larger circuits ($p \geq 11$). Even though ILA$_{(tfhe)}$'s runtime increased with more number of computations, the rate of increase is much slower than that of TFHE-rs. This is unsurprising as FHE circuits are known to be slow, and any dynamic detection inherits the poor performance. Though a GPU support might accelerate the performance of TFHE-rs, ILA$_{(tfhe)}$ may still outperform as it does not execute the FHE circuit for overflow detection.

Modulus Switch inference. Our modulus switch inference algorithm is an optimization that can further tighten the multiplicative depth that ILA can guarantee. We implemented the proposed algorithm using the Microsoft SEAL library as the backend, and evaluated it across multiple modulus chains ordered. We sorted elements of each modulus chain in decreasing order. Experimental results show that, in almost all tested cases, the multiplicative depth inferred by our MS algorithm matches the maximum achievable depth for the corresponding modulus chain.

9 Related Work

Existing FHE Solutions. Table 3 summarizes the support for correctness reasoning among the state-of-the-art FHE libraries. Only a few libraries support more than one FHE scheme and all of them have limited—if any—error detection support. Neither OpenFHE [5], SEAL [22], Lattigo [4] or HElib [16] handle value overflows. As a concrete example, if the plaintext modulus is 100, and the decrypted output is 20, then the intended result could be one of 120, −20 or 50720.

SEAL, Lattigo, HElib and OpenFHE [5] support BGV and BFV schemes including modulus switching operation. None of them support modulus switch inference. OpenFHE and TFHE-rs [24] also support TFHE although the former's support is limited in functionality.

Noise overflow issues differ with every library. OpenFHE guarantees are valid up to a preselected depth (chosen by the user) of cipher-cipher operations; however, the guarantees get invalidated in the presence of other homomorphic operations (e.g., cipher-plain multiplication). To overcome this limitation, OpenFHE recommends

FHE Library	Schemes	O_{value}	O_{noise}	SK Ind	Static
OpenFHE [5]	✓	✗	✗	✗	✗
SEAL [22]	✗	✗	✗	✗	✗
Lattigo [4]	✓	✗	✗	✗	✗
HElib [16]	✗	✗	✓	✓	✗
TFHE-rs [24]	✗	✓	✗	✓	✗
ILA (this work)	✓	✓	✓	✓	✓

Table 3: The first column indicates support for multiple schemes. The next two columns check for value and noise overflow errors. The fourth column verifies secret key independence, and the final column indicates whether the technique is static.

estimating the depth parameter that over-approximates all operations as cipher-cipher [2]. Since noise growth is slower for other combinations, the over-approximation leads to a much slower circuit (due to larger parameters). Moreover, it is tricky to get these estimations correct as the depth is a function of fresh ciphertext operations whereas the programs often operate on a series of mutated ciphertexts.

SEAL and Lattigo do not offer any noise estimation whereas HElib supports explicit user queries on noise estimate queries at runtime. Note that the latter setup is undesirable for two reasons. First, a user has to explicitly query whether an overflow has occurred. Second, it can be orders of magnitude slower as FHE circuits are well-known to suffer from poor performance.

TFHE-rs is more nuanced: it supports detecting value overflows but not noise overflows. However, value overflow detection requires running the FHE circuit which is once again slow. In this work, we use a static approach that can be orders of magnitude fast (Section 8). Interestingly, it has a noise management strategy that automatically inserts PBS operations after a preselected operation depth (controlled by the user). Hence, noise overflow might appear to be of lesser concern. However, we argue that it isn't. First, the user may choose to turn it off for some operations (e.g., addition). Second, the operation depth for PBS is chosen apriori and is agnostic to whether a PBS is required. As PBS is computationally expensive, this approach is sub-optimal and further slows down the circuit. More generally, precise PBS insertion would entail noise overflow detection that TFHE-rs lacks.

Other FHE Compilers. We refer the reader to a survey on the state-of-the-art in FHE compilers [23]. To the best of our knowledge, ILA is the only work to prove both functional correctness property. Alchemy [14] is an embedded DSL in Haskell for using FHE with type-level tracking of ciphertext noise; however, neither do these type-level noise computations come with formal guarantees, nor are the heuristics for noise estimations sound.

Other FHE compilers such as EVA [15], Coyote [20] and others [10, 14, 23] focus on efficient encoding schemes but not formal guarantees. They are orthogonal to our work.

10 Conclusion and Future Work

We present ILA, a correctness-oriented IR and an abstract model for FHE circuits. Our IR is backed by a type system that statically tracks low-level quantitative bounds (e.g., ciphertext noise) without using the secret key. Using our type system, we identify and prove a strong *functional correctness* criterion for ILA circuits. Furthermore, we instantiate ILA with three absolute schemes—BGV, BFV and TFHE—and get functional correctness for free. Comparative evaluation of ILA against three popular FHE libraries shows that ILA's static analysis is sound, tight and efficient.

In future work, we aim to extend ILA to support functional, higher-order programming with FHE. While function types are straightforward, they are not ergonomic without *bound polymorphism*, which would enable types such as $\forall\alpha.\ \text{cipher}\ \alpha \rightarrow \text{cipher}\ (2*\alpha)$. Bound polymorphism in our setting requires extending the noise inference algorithms of the underlying cryptosystems (e.g., BGV) to handle *symbolic* bounds, which will likely require insights from nonlinear arithmetic constraint solving.

Additionally, we aim to instantiate ILA with approximate FHE schemes such as CKKS [9]. This extension primarily requires our notion of an ILA model to be generalized so that Commutativity does not hold on the nose, but holds with some bounded amount of error.

References

[1] Private Information Retrieval . URL https://swiftpackageindex.com/apple/swift-homomorphic-encryption/1.0.4/documentation/privateinformationretrieval/parametertuning.

[2] Noise growth in ciphertext-plaintext multiplication. URL https://openfhe.discourse.group/t/multiplication-depth-in-bfv/487/7.

[3] Coefficient modulus bit count in microsoft seal. URL https://github.com/microsoft/SEAL/issues/232#issuecomment-711103657.

[4] Lattigo v6. Online: https://github.com/tuneinsight/lattigo, August 2024. EPFL-LDS, Tune Insight SA.

[5] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3560827.3563379. URL https://doi.org/10.1145/3560827.3563379.

[6] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits. In *Topics in Cryptology – CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*, page 345–363, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-40185-6. doi: 10.1007/978-3-030-40186-3_15. URL https://doi.org/10.1007/978-3-030-40186-3_15.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311151. doi: 10.1145/2090236.2090262.

[8] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. Cryptology ePrint Archive, Paper 2017/299, 2017. URL https://eprint.iacr.org/2017/299.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70694-8.

[10] 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*, 2018.

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421, 2018. URL https://eprint.iacr.org/2018/421.

[12] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Proceedings of the RSA Conference on Topics in Cryptology - CT-RSA 2016 - Volume 9610*, page 325–340, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 9783319294841. doi: 10.1007/978-3-319-29485-8_19. URL https://doi.org/10.1007/978-3-319-29485-8_19.

[13] Anamaria Costache, Kim Laine, and Rachel Player. Evaluating the effectiveness of heuristic worst-case noise analysis in fhe. In *Computer Security – ESORICS*

2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II, page 546–565, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-59012-3. doi: 10.1007/978-3-030-59013-0_27. URL https://doi-org/10.1007/978-3-030-59013-0_27.

[14] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1020–1037, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243828. URL https://doi.org/10.1145/3243734.324382.

[15] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 546–561, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386023. URL https://doi.org/10.1145/3385412.3386023.

[16] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44371-2.

[17] Ilia Iliashenko. Optimisations of fully homomorphic encryption. 2019.

[18] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi: 10.1145/3656382. URL https://doi.org/10.1145/3656382.

[19] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, page 648–677, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-77869-9. doi: 10.1007/978-3-030-77870-5_23. URL https://doi-org/10.1007/978-3-030-77870-5_23.

[20] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 118–133, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582057. URL https://doi.org/10.1145/3582016.3582057.

[21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480070. URL https://doi.org/10.1145/3466752.3480070.

[22] SEAL. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[23] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108, 2021. doi: 10.1109/SP40001.2021.00068.

[24] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[25] Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Ximeng Liu, and Kai Chen. Sok: Fully homomorphic encryption accelerators. *ACM Comput. Surv.*, 56(12), October 2024. ISSN 0360-0300. doi: 10.1145/3676955. URL https://doi.org/10.1145/3676955.

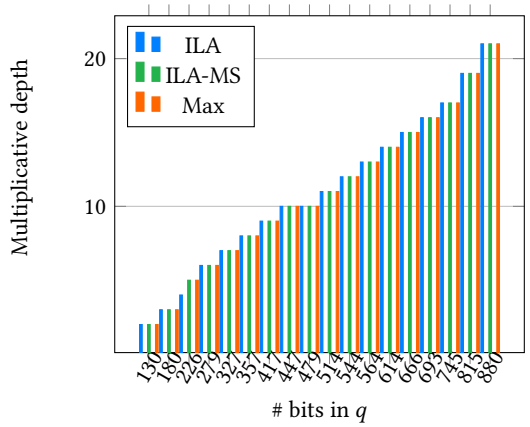

Figure 17: Multiplicative depth before and after modulus switch inference.

A Modulus Switching

Since the focus of modulus switch inference is to demonstrate the role of the type system in transformation validation, we keep it simple. Our inference algorithm works for loop-free programs. Multiplicative depth computation for ciphertexts inside a loop depends on the loop iteration. One potential way to fix this would be to carry out the inference after unrolling the loop. Since ILA programs only have finite loops, this is feasible.

As an example, suppose that the last iteration of the PSI program (Figure 2) overflows. Then this approach would yield a program where lines 14 – 16 in the unrolled version of the program have modulus switching operations as shown below.

```
...                  # unroll the inner loop
t5 = A[0]            # last iteration
t6 = modswitch(t5) ⊕ modswitch(t2) # switch levels
t7 = modswitch(result) ⊗ t6    # level propagation
result = t7 ⊗ modswitch(t4)
```

B Evaluation

Evaluation shows comparative multiplicative depth of ILA against SEAL and OpenFHE for BGV and BFV schemes. Figure **??** compares ILA$_{(bgv)}$ with SEAL whereas Figures 18, 19, 20 and 21 compare ILA with OpenFHE for cipher-cipher and cipher-plain multiplications. Additionally, Figure 17 compares the multiplicative depth before and after modulus switching inference.

C Proofs

Theorem C.1 (Semantic Safety: Expressions). *Suppose that* topub(sp); $\Gamma \vdash e : \tau$. *Then* sp; $\Gamma \vDash e : \tau$.

Proof. Recall that we must show that, for all γ such that sp; $\Gamma \vDash \gamma$ (i.e., $\gamma(x) \in [\![\Gamma(x)]\!]^{\text{sp}}$ for all $x \in \Gamma$), there exists a v such that $\langle \gamma, e \rangle \Downarrow v$ and $v \in [\![\tau]\!]^{\text{sp}}$.

By induction on the typing derivation (Figure 8).

- Case var: We have that $\langle \gamma, x \rangle \Downarrow \gamma(x)$, $\gamma(x) \in [\![\Gamma(x)]\!]^{\text{sp}}$, and $\tau = \Gamma(x)$. The result follows.
- Case sub: The result follows once we show that, for all sorts $s \in \{\text{cipher}, \text{plain}, \text{msg}\}$, if $\alpha \leq_s \alpha'$, then $[\![s\ \alpha]\!]^{\text{sp}} \subseteq [\![s\ \alpha']\!]$.

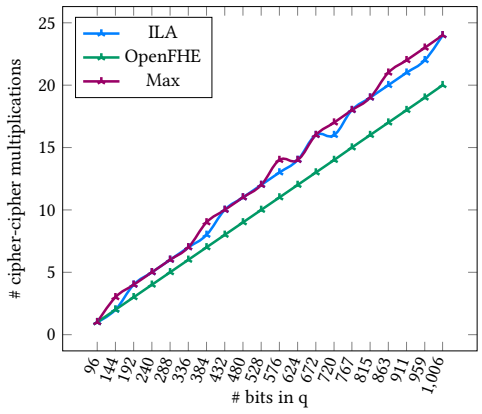

Figure 18: Multiplicative depth of ILA$_{(bgv)}$ vs OpenFHE. X-axis shows the number of bits in coefficient modulus q; Y-axis shows number of plain-cipher multiplications estimated by ILA, OpenFHE vs maximal possible (higher is better).

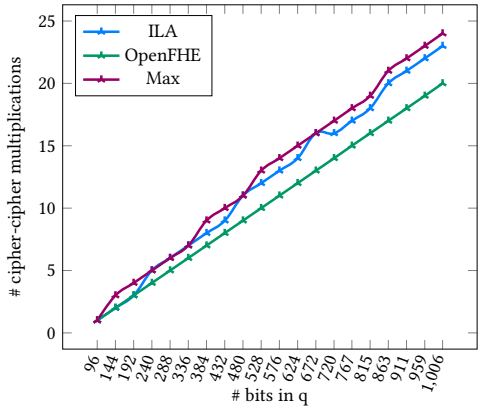

Figure 19: Multiplicative depth of ILA$_{(bfv)}$ vs OpenFHE. X-axis shows the number of bits in coefficient modulus q; Y-axis shows number of plain-cipher multiplications estimated by ILA, OpenFHE vs maximal possible (higher is better).

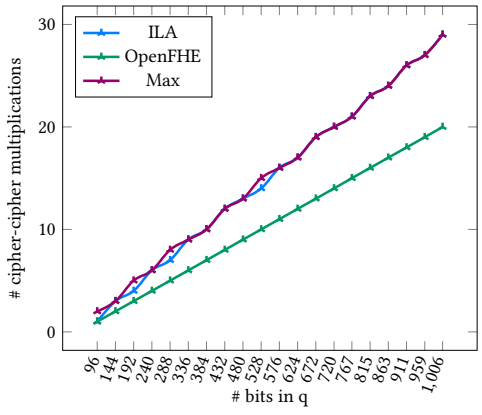

Figure 20: Plain-cipher multiplicative depth of ILA$_{(bgv)}$ vs OpenFHE.

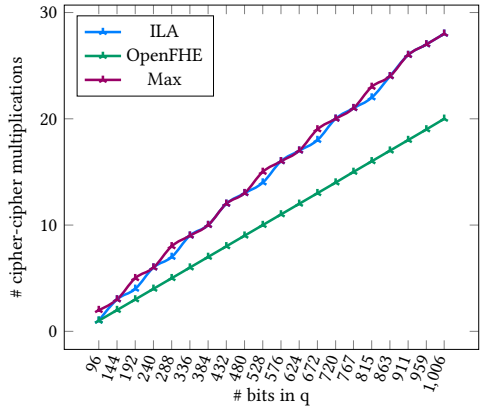

Figure 21: Plain-cipher multiplicative depth of ILA$_{(bfv)}$ vs OpenFHE. X-axis shows the number of bits in coefficient modulus q; Y-axis shows number of plain-cipher multiplications estimated by ILA, OpenFHE vs maximal possible (higher is better).

Since $[\![s\ \alpha]\!]^{\mathsf{sp}} = \{v \in [\![s]\!] \mid |v|_s^{\mathsf{sp}} \leq_s \alpha\}$, this follows from transitivity of $\leq_s$.

- Case Const: We need to show that, for $s \in \{\mathsf{msg}, \mathsf{plain}\}$, for all $v \in [\![s]\!]$, $v \in [\![s\ |v|_s^{\mathsf{topub(sp)}}]\!]^{\mathsf{sp}}$. Hence, we must show that $|v|_s^{\mathsf{topub(sp)}} \leq_s |v|_s^{\mathsf{sp}}$. For $s \in \{\mathsf{msg}, \mathsf{plain}\}$, the right bound is defined to be the left.
- Case op: We know that for all i, $\mathsf{sp}; \Gamma \vDash e_i : \tau_i$ and that $[\![op]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|\tau_i|}) = \alpha$, and we must prove that $\mathsf{sp}; \Gamma \vDash \mathsf{op}(\overrightarrow{e_i}) : s\ \alpha$.
  To this end, take γ such that $\mathsf{sp}; \Gamma \vDash \gamma$. By induction, we have that $\langle \gamma, e_i \rangle \Downarrow v_i \in [\![\tau_i]\!]^{\mathsf{sp}}$ for all i, and thus $\langle \gamma, \mathsf{op}(\overrightarrow{e_i}) \rangle \Downarrow v = [\![\mathsf{op}]\!](\overrightarrow{v_i})$. We know that $\mathsf{sort}(v) = s$. We must show that $|[\![\mathsf{op}]\!](\overrightarrow{v_i})|_s^{\mathsf{sp}} \leq \alpha$. First, observe that $[\![\mathsf{op}]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|v_i|_{\mathsf{sort}(\tau_i)}^{\mathsf{sp}}}) \leq [\![\mathsf{op}]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|\tau_i|}) = \alpha$ by the Downwards Closed axiom. Then, by the Commutativity axiom, we have $|[\![\mathsf{op}]\!](\overrightarrow{v_i})_s^{\mathsf{sp}}| \leq_s \alpha$.

□

Theorem C.2 (Semantic Safety: Commands). *Suppose that* $\mathsf{topub(sp)}; \Gamma \vdash c \dashv \Gamma'$. *Then* $\mathsf{sp}; \Gamma \vDash c \dashv \Gamma'$.

Proof. Recall that we need to show that whenever $\mathsf{sp}; \Gamma \vDash \gamma$ and $\langle \gamma, c \rangle \Downarrow \gamma'$, that $\mathsf{sp}; \Gamma' \vDash \gamma'$.

Induct on the typing derivation of c (Figure 9):

- Case skip: immediate, since $\langle \gamma, \mathsf{skip} \rangle \Downarrow \gamma$.
- Case Assgn: we have that $\langle \gamma, x := e \rangle \Downarrow \gamma[x \mapsto v]$, where $\langle \gamma, e \rangle \Downarrow v$, and we must show that $\mathsf{sp}; \Gamma[x \mapsto \tau] \vDash \gamma[x \mapsto v]$. By Theorem C.1, we have that $v \in [\![\tau]\!] \in [\![\tau]\!]^{\mathsf{sp}}$. Since we are updating Γ to always point to τ, the result follows.
- Case seq: immediate, since we know by induction that any post-state of c_1 will satisfy its output type context, which is assumed to be the input type context of c_2.
- Case if: we have that $\langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rangle$ evaluates as $\langle \gamma, c_1 \rangle \Downarrow \gamma_1$ if $\langle \gamma, e \rangle$ evaluates to $[\![true]\!]()$, and $\langle \gamma, c_2 \rangle \Downarrow \gamma_2$ otherwise. By induction, we have that $\mathsf{sp}; \Gamma_1 \vDash \gamma_1$ and $\mathsf{sp}; \Gamma_2 \vDash \gamma_2$. We must show that if $\Gamma_1 \sqcap \Gamma_2 \sim \Gamma'$, then $\mathsf{sp}; \Gamma' \vDash \gamma_1$ and $\mathsf{sp}; \Gamma' \vDash \gamma_2$.
  Take $x \in \Gamma'$. Then, by assumption, $x \in \Gamma_1$ and $\Gamma_1(x) \leq \Gamma'(x)$. Since $\gamma_1(x) \in [\![\Gamma_1]\!]^{\mathsf{sp}}$, we have that $\gamma_1(x) \in [\![\Gamma']\!]^{\mathsf{sp}}$ since subtyping respects $[\![\cdot]\!]^{\mathsf{sp}}$. The argument is the same for Γ_2.

□

Theorem C.3 (Message Equivalence: Expressions). *Suppose that* $\mathsf{topub(sp)}; \Gamma \vdash e : \tau$ *and* $\mathsf{sp}; \Gamma \vDash \gamma$. *Let*

$$\gamma' = \{x \mapsto \mathsf{interp}_{\mathsf{sort}(\Gamma(x))}^{\mathsf{sp}}(\gamma(x))\}.$$

Then, $\langle \gamma, e \rangle \Downarrow v$ *and* $\langle \gamma', e \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} v_{\mathsf{msg}}$ *such that* $\mathsf{interp}_{\mathsf{sort}(v)}^{\mathsf{sp}}(v) = v_{\mathsf{msg}}$.

Proof. By induction on the typing derivation.

- Case var: we have that if $e = x$, then $v = \gamma(x)$ and $v_{\mathsf{msg}} = \gamma'(x) = \mathsf{interp}_{\mathsf{sort}(\Gamma(x))}^{\mathsf{sp}}(\gamma(x))$. The result follows from the fact that $\mathsf{sort}(\gamma(x)) = \mathsf{sort}(\Gamma(x))$, since $\mathsf{sp}; \Gamma \vDash \gamma$.
- Case sub: the inductive hypothesis on e is equivalent to the result, since the sort of τ does not change (only the bound).
- Case const: if $e = v \in [\![s]\!]$, $\langle \gamma, v \rangle \Downarrow v$ and $\langle \gamma', v \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} \mathsf{interp}_s^{\mathsf{sp}}(v)$. The result follows.
- Case op: We have that $e = \mathsf{op}(\overrightarrow{e_i})$ where $\mathsf{op} : \overrightarrow{s_i} \to s$, and $[\![op]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|\tau_i|})$ is defined. Inductively, we have that for each i, $\langle \gamma, e_i \rangle \Downarrow v_i$ and $\langle \gamma', e_i \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} v_{i,\mathsf{msg}}$ such that $\mathsf{interp}_{s_i}^{\mathsf{sp}}(v_i) = v_{i,\mathsf{msg}}$. We also have that $\langle \gamma, \mathsf{op}(\overrightarrow{e_i}) \rangle \Downarrow [\![op]\!](\overrightarrow{v_i})$ and that $\langle \gamma', \mathsf{op}(\overrightarrow{e_i}) \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} [\![op]\!]_{\mathsf{msg}}(\overrightarrow{v_{i,\mathsf{msg}}})$. Also, observe that $[\![\mathsf{op}]\!](\overrightarrow{v_i}) \in [\![s]\!]$ by assumption.
  Putting everything together, we need to show that

$$\mathsf{interp}_s^{\mathsf{sp}}([\![op]\!](\overrightarrow{v_i})) = [\![\mathsf{op}]\!]_{\mathsf{msg}}(\overrightarrow{\mathsf{interp}_{s_i}^{\mathsf{sp}}(v_i)}).$$

  This is the conclusion of Commutativity in the model.
  To apply Commutativity, we must show that

$$[\![\mathsf{op}]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|v_i|_{s_i}^{\mathsf{sp}}})$$

  is defined. From Semantic Safety, we have that $v_i \in [\![\tau_i]\!]^{\mathsf{sp}}$ for all i, and hence $|v_i|_{s_i}^{\mathsf{sp}} \leq_{s_i} |\tau_i|$. Since $[\![op]\!]_{\mathsf{bnd}}^{\mathsf{topub(sp)}}(\overrightarrow{|\tau_i|})$ is defined, the result follows from the Downwards Closed axiom in the model.

□

Theorem C.4 (Message Equivalence: Commands). *Suppose that* $\mathsf{topub(sp)}; \Gamma \vdash c \dashv \Gamma'$ *and* $\mathsf{sp}; \Gamma \vDash \gamma$. *Let* $\gamma' = \{x \mapsto \mathsf{interp}_{\mathsf{sort}(\Gamma(x))}^{\mathsf{sp}}(\gamma(x))\}$. *Then, if* $\langle \gamma, c \rangle \Downarrow \gamma_1$, *we have that* $\langle \gamma', c \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} \gamma_1'$ *such that, for all* $x \in \gamma_1$, $\mathsf{interp}_{\mathsf{sort}(\gamma_1(x))}^{\mathsf{sp}}(\gamma_1(x)) = \gamma_1'(x)$.

Proof. Follows by induction on the typing derivation.

- Case skip: We have that $\langle \gamma, \mathsf{skip} \rangle \Downarrow \gamma$ and $\langle \gamma', \mathsf{skip} \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} \gamma'$. The result follows by assumption.
- Case Assgn: We have that $\langle \gamma, y := e \rangle \Downarrow \gamma[y \mapsto v]$ where $\langle \gamma, e \rangle \Downarrow v$, and $\langle \gamma', y := e \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} \gamma'[y \mapsto v']$ where $\langle \gamma', e \rangle \Downarrow_{\mathsf{msg}}^{\mathsf{sp}} v'$.

If $x \neq y$, the result follows by assumption. If $x = y$, the result follows by Message Equivalence on expressions (Theorem C.3).

- Case SEQ: The result follows, since the postcondition of the inductive hypothesis for c_1 matches the precondition of the inductive hypothesis for c_2.
- Case IF: Note that since the guard e of the if statement has type msg α, and $\mathsf{interp}^{\mathsf{sp}}_{\mathsf{msg}}(v) = v$ for all v, we will have that if e then c_1 else c_2 will evaluate in the same direction for both γ and γ'. If e evaluates to $[\![true]\!]()$, then the result follows by the inductive hypothesis on c_1; otherwise, the result follows by the inductive hypothesis on c_2.

□

D ILA$_{(tfhe)}$ instantiation

We introduce ILA$_{(tfhe)}$, an instantiation of ILA model for the CGGI/TFHE scheme. TFHE is a fast bootstrapping GLWE scheme.

Definition D.1 shows the full instantiation. Note that it closely follows the definition of ILA model (see Definition 4.1, Section 4).

DEFINITION D.1 (ILA$_{(tfhe)}$). *The ILA$_{(bgv)}$ model is defined as follows:*

- *Bounds sets:*
  - $B_{\mathsf{msg}} = \emptyset$;
  - $B_{\mathsf{plain}} = \{(\mathsf{inf}, \mathsf{sup}, \epsilon) \mid \mathsf{inf} \leq \mathsf{sup}$ *and* $\epsilon \in \mathbb{N}\}$ *ordered by* $\leq_{\mathsf{plain}}$;
  - $B_{\mathsf{cipher}} = \{(\mathsf{id}, \mathsf{inf}, \mathsf{sup}, \epsilon) \mid \mathsf{inf} \leq \mathsf{sup}, \mathsf{id} \in \{\mathsf{LWE}, \mathsf{RLWE}, \mathsf{RGSW}\}$ *and* $\epsilon \in \mathbb{N}\}$ *ordered by* $(\mathsf{id}, \mathsf{inf}_1, \mathsf{sup}_1, \epsilon_1) \leq_{\mathsf{cipher}} (\mathsf{id}, \mathsf{inf}_2, \mathsf{sup}_2, \epsilon_2)$ *if* $\mathsf{inf}_2 \leq \mathsf{inf}_1 \leq \mathsf{sup}_1 \leq sup_2$ *and* $\epsilon_1 \leq \epsilon_2$;
- *Mappings for bounds computation:*
  - $|m|^{\mathsf{pp}}_{\mathsf{msg}} = \emptyset$;
  - $|p|^{\mathsf{pp}}_{\mathsf{plain}} = (\mathsf{inf}, \mathsf{sup})$; *computed as in ILA$_{(bgv)}$*
  - $|\mathsf{ct}|^{\mathsf{sp}}_{\mathsf{cipher}} = (\mathsf{id}, \mathsf{inf}, \mathsf{sup}, \epsilon)$ *computation similar to ILA$_{(bgv)}$*
- *Decoding on plaintexts:* $\mathsf{interp}^{\mathsf{pp}}_{\mathsf{plain}}(p) = \mathsf{decode}(p)$;
- *Decryption on ciphertexts:* $\mathsf{interp}^{\mathsf{sp}}_{\mathsf{cipher}}(\mathsf{ct}) = \mathsf{decode}(\mathsf{decrypt}(\mathsf{ct}))$;
- *Operator* $\mathsf{op} \in \{\oplus, \otimes, \times, \mathsf{cmux}, \boxtimes, \boxdot, \mathsf{pbs}\}$ *defined as follows:*
  - $[\![\otimes]\!]$: {cipher, plain}, {cipher, plain} → {cipher, plain};
  - $[\![\oplus]\!]$: {cipher, plain}, {cipher, plain} → {cipher, plain};
  - $[\![\mathsf{cmux}]\!]$: {cipher, cipher, cipher} → {cipher};
  - $[\![\boxdot]\!]$: {cipher, cipher} → {cipher};
  - $[\![\mathsf{pbs}]\!]$: {cipher, cipher, cipher} → {cipher};
  - $[\![\times]\!]$: int, cipher → cipher; *and*
  - *bounds manipulation mapping* $[\![\mathsf{op}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}$ *shown in Table 4.*

We prefix 'cipher' with id whenever id is significant. For simplicity, we assume that LWE < RLWE < RGSW.

D.0.1 Instantiation of ILA Functional Correctness. The commutativity and downwards closure axioms are similar to that of ILA$_{(bgv)}$. Since ILA$_{(tfhe)}$ is a valid ILA model, we thus get Theorem 4.4 for free.

Sorts	s	$\in$	{msg, plain, cipher}
Bounds	α_s	$\in$	B_s
Types	τ	::=	cipher α_{cipher} | plain α_{plain} | msg α_{msg}
Contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$

Figure 22: ILA Extensions

$$\boxed{\mathsf{pp}; \Gamma \vdash e : \tau} \qquad \frac{x : \tau \in \Gamma}{\mathsf{pp}; \Gamma \vdash x : \tau} \qquad \frac{\mathsf{sort}(v) = s}{\mathsf{pp}; \Gamma \vdash v : s\ |v|^{\mathsf{pp}}_{s}}$$

$$\frac{\mathsf{pp}; \Gamma \vdash e_i : s\ \alpha_i}{\mathsf{pp}; \Gamma \vdash [e_1, \ldots, e_n] : \mathsf{vec}\ s\ (\alpha_1, \ldots, \alpha_n)}$$

$$\frac{\mathsf{op} : \overrightarrow{s_i} \to s \qquad (\forall i, \mathsf{pp}; \Gamma \vdash e_i : \tau_i \qquad \mathsf{sort}(\tau_i) = s_i) \qquad [\![\mathsf{op}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(|\overrightarrow{\tau_i}|) = \alpha}{\mathsf{pp}; \Gamma \vdash \mathsf{op}(\overrightarrow{e_i}) : s\ \alpha}$$

Figure 23: ILA Typing: Expression including vectors

E ILA$_{(bgv)}$ modswitch Typing Rule

As yet another example, we show the typing rule for modswitch operation using the bounds computation from Table 1. Recall that modulus switching reduces both the modulus level and noise on a ciphertext. Rule T-MS states that modulus switching reduces the noise by $\frac{q_{\omega-1}}{q_\omega}$ with some rounding error B_r. Additionally, the level of the ciphertext is lowered from ω to $\omega-1$. Notice that the resulting noise ϵ' is always lower than ϵ due to decreasing chain property of modulus levels.

T-MS

$$\frac{\mathsf{pp}; \Gamma \vdash e : \mathsf{cipher}\ \langle \mathsf{inf}, \mathsf{sup}, \epsilon, \omega \rangle \qquad 0 \leq \omega - 1 < L \qquad \epsilon' = \frac{q_{\omega-1}}{q_\omega}\epsilon + B_r \leq l_{\omega-1}}{\mathsf{pp}; \Gamma \vdash \mathsf{modswitch}(e) : \mathsf{cipher}\ \langle \mathsf{inf}, \mathsf{sup}, \epsilon', \omega - 1 \rangle}$$

F ILA Extensions

Op	$[\![\cdot]\!]^{\mathsf{pp}}_{\mathsf{bnd}}$	Conditions
$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{id}_i, \mathsf{inf}_i, \mathsf{sup}_i, \epsilon_i})$	$= (\mathsf{id}, \mathsf{inf}, \mathsf{sup}, \epsilon)$	$\mathsf{id}_i \in \{\mathsf{LWE}, \mathsf{RLWE}\}$ and $\mathsf{id} = \max\{\mathsf{LWE}, \mathsf{RLWE}\}$
$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{inf}_i, \mathsf{sup}_i})$	$= (\mathsf{inf}, \mathsf{sup})$	
$[\![\oplus]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\mathsf{inf}_1, \mathsf{sup}_1), (\mathsf{id}, \mathsf{inf}_2, \mathsf{sup}_2, \epsilon)$	$= (\mathsf{id}, \mathsf{inf}, \mathsf{sup}, \epsilon)$	
where		and in all cases
$\mathsf{inf} := \mathsf{inf}_1 + \mathsf{inf}_2$, $\mathsf{sup} := \mathsf{sup}_1 + \mathsf{sup}_2$, $\epsilon := \epsilon_1 + \epsilon_2$		$-t/2 \le \mathsf{inf} \le \mathsf{sup} < t/2$ and $\epsilon \le q/2t$
$[\![\otimes]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{inf}_i, \mathsf{sup}_i})$	$= (\mathsf{inf}, \mathsf{sup})$	
$[\![\boxtimes]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\overrightarrow{\mathsf{RGSW}, \mathsf{inf}_i, \mathsf{sup}_i, \epsilon_i})$	$= (\mathsf{RGSW}, \mathsf{inf}, \mathsf{sup}, f'(\epsilon_1, \epsilon_2))$	$f'(\epsilon_1, \epsilon_2) \le q/t$
$[\![\boxdot]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\mathsf{RGSW}, \mathsf{inf}_1, \mathsf{sup}_1, \epsilon_1), (\mathsf{LWE}, \mathsf{inf}_2, \mathsf{sup}_2, \epsilon_2)$	$= (\mathsf{RLWE}, \mathsf{inf}, \mathsf{sup}, g(\epsilon_1, \epsilon_2))$	$g(\epsilon_1, \epsilon_2) \le q/t$
$[\![\boxdot]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\mathsf{RGSW}, \mathsf{inf}_1, \mathsf{sup}_1, \epsilon_1), (\mathsf{RLWE}, \mathsf{inf}_2, \mathsf{sup}_2, \epsilon_2)$	$= (\mathsf{RLWE}, \mathsf{inf}, \mathsf{sup}, g'(\epsilon_1, \epsilon_2))$	$g'(\epsilon_1, \epsilon_2) \le q/t$
		where
		$\mathsf{inf} := \min\{\mathsf{inf}_1 * \mathsf{inf}_2, \mathsf{inf}_1 * \mathsf{sup}_2, \mathsf{sup}_1 * \mathsf{inf}_2, \mathsf{sup}_1 * \mathsf{sup}_2\}$
		$\mathsf{sup} := \max\{\mathsf{inf}_1 * \mathsf{inf}_2, \mathsf{inf}_1 * \mathsf{sup}_2, \mathsf{sup}_1 * \mathsf{inf}_2, \mathsf{sup}_1 * \mathsf{sup}_2\}$
$[\![\times]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(n, (\mathsf{id}, \mathsf{inf}, \mathsf{sup}, \epsilon))$	$= (\mathsf{id}, \mathsf{inf}', \mathsf{sup}', \lvert n * \epsilon \rvert)$	$\mathsf{inf}' = \min\{n * \mathsf{inf}, n * \mathsf{sup}\}$ and $\mathsf{sup}' = \max\{n * \mathsf{inf}, n * \mathsf{sup}\}$
		if $-t/2 \le \mathsf{inf}' \le \mathsf{sup}' < t/2$ and $\lvert n * \epsilon \rvert \le q/t$
$[\![\mathsf{pbs}]\!]^{\mathsf{pp}}_{\mathsf{bnd}}(\mathsf{RGSW}, \mathsf{inf}_0, \mathsf{sup}_0, \epsilon_0)(\mathsf{LWE}, \mathsf{inf}, \mathsf{sup}, \epsilon)$	$= (\mathsf{LWE}, \mathsf{inf}, \mathsf{sup}, \epsilon_b)$	if $\max\{\log_2 \lvert \mathsf{inf} \rvert, \log_2 \lvert \mathsf{sup} \rvert\} \le (\log_2(t) - 1)$
		if $-t/2 \le \mathsf{inf}_0 \le \mathsf{sup}_0 < t/2$

Table 4: ILA$_{(tfhe)}$: ILA model for TFHE scheme illustrating key homomorphic operations. Note that input operand has the type $(id_i, \mathsf{inf}_i, \mathsf{sup}_i, \epsilon_i)$ (for ciphertext)