Towards A Formally Verified Fully Homomorphic Encryption Compute Engine *

Jeremy Casas[1], Zhenkun Yang[1], Wen Wang[1], Jin Yang[1], Adwait Godbole[2]
[1]*Intel Labs, Intel Corporation*, [2]*University of California, Berkeley*
jeremy.casas@intel.com, zhenkun.yang@intel.com, wen.wang@intel.com, jin.yang@intel.com, adwait@berkeley.edu

***Abstract*—We present a scalable approach for formally verifying the correctness of the Compute Engine (CE) against its ISA (Instruction Set Architecture) specification in an FHE (Fully Homomorphic Encryption) accelerator, critical to many applications where safety and security of information is of vital importance. It combines algorithmic verification of the micro-architecture modules in the CE against their functional specifications and implementation verification of the CE hardware against its micro-architecture algorithmic specifications. The correctness of the CE is guaranteed by treating micro-architecture modules as semantic-preserving program transformations and leveraging the composability of the semantic-preserving properties well established in compiler design and verification.**

I. Introduction

The safety and security of critical information – whether it is sensitive intellectual property, financial information, personally identifiable information, intelligence insight, or beyond – is of vital importance. Current methods protect data as it is transmitted across a network, at rest, or while in storage. Processing or computing on this data, however, requires that it is first decrypted, exposing it to numerous vulnerabilities and threats. Fully homomorphic encryption (FHE) [1] offers a solution to this challenge by enabling computation on encrypted data to keep data protected at all times.

Despite its potential, FHE requires enormous computation time to perform even simple operations, making it exceedingly impractical to implement with traditional hardware. For instance, the training of a 7-layer convolutional neural network that takes less than an hour for unencrypted data will take years to complete on encrypted data on the same processor [2].

Several research efforts are underway to accelerate the FHE computations in response to the DARPA's challenges in the Data Protection in Virtual Environments (DPRIVE) program [2]. One primary goal is to dramatically accelerate FHE calculations to within one order of magnitude of current performance on unencrypted data. Fig. 1 shows the high-level architecture of our FHE accelerator, which is partitioned into one or more compute engines (CE) with a shared cache to execute SIMD (Single Instruction Multiple Data) polynomial ring (poly-ring) instructions in parallel on $8k$ bufferfly compute elements across 128 tiles, and a compiler that translates a stream of FHE operations into a semantically-equivalent stream of SIMD poly-ring instructions.

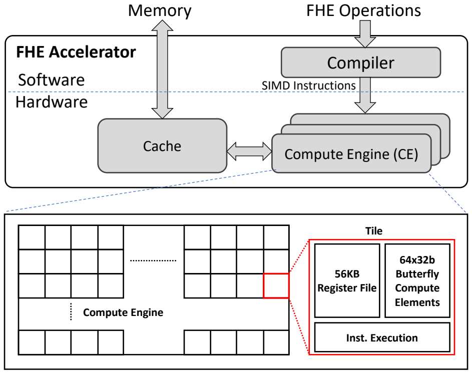


Fig. 1. FHE Accelerator Architecture

With the wide range of critical applications FHE intends to support, it is important to formally verify the functional correctness of the FHE accelerator, as any bugs in the accelerator can undermine the entire system integrity and result in losses of billions of dollars financially or even catastrophic effects on human lives or infrastructure. Although formal verification has been long promised as a solution to ensure full correctness of a design, existing approaches have fallen far short to be practical for the end-to-end correctness of a complex accelerator design due to capacity limitation. A-QED [3] proposed *bounded model checking* based approaches for verification of hardware accelerators, which only handles relatively small designs with partially captured specifications. There are also attempts to verify optimized Number Theoretic Transform (NTT) algorithms [4] using bounded model checking and abstract interpretation. It mainly focuses on numerical overflows at the algorithm level (C/C++).

We developed a framework for scalable accelerator design and verification, inspired by composable compiler design and verification techniques, by treating micro-architecture modules of an accelerator as meta-programs to perform semantic-preserving program transformations. In this paper, we show how to apply this framework to the verification of the FHE accelerator, focusing on the CE hardware and show an example on how to extend it to the compiler stack.

II. Overview of the Verification Approach

The end-to-end execution of the FHE accelerator starts from a sequence of FHE operations that are converted into equivalent but more hardware friendly micro-operations (*e.g.*, via a compiler stack), and ultimately executes on the CE hardware. Fig. 2 shows six meta-program modules that perform a

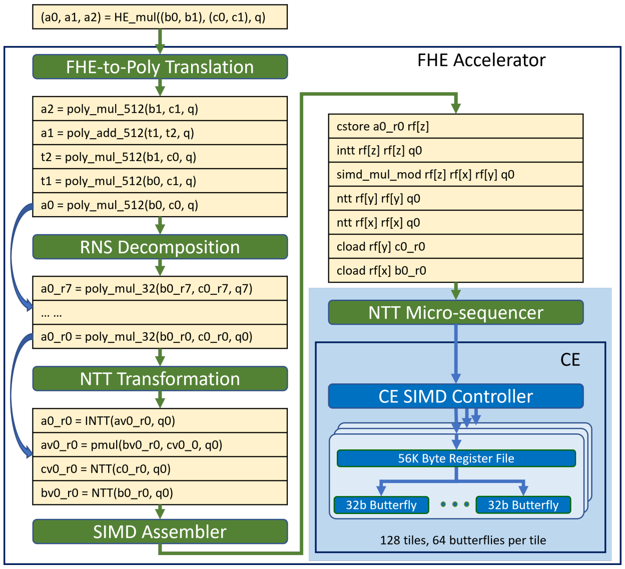

(a0, a1, a2) = HE_mul((b0, b1), (c0, c1), q)
FHE Accelerator
FHE-to-Poly Translation
a2 = poly_mul_512(b1, c1, q)
a1 = poly_add_512(t1, t2, q)
t2 = poly_mul_512(b1, c0, q)
t1 = poly_mul_512(b0, c1, q)
a0 = poly_mul_512(b0, c0, q)
RNS Decomposition
a0_r7 = poly_mul_32(b0_r7, c0_r7, q7)
… …
a0_r0 = poly_mul_32(b0_r0, c0_r0, q0)
NTT Transformation
a0_r0 = INTT(av0_r0, q0)
av0_r0 = pmul(bv0_r0, cv0_0, q0)
cv0_r0 = NTT(c0_r0, q0)
bv0_r0 = NTT(b0_r0, q0)
SIMD Assembler
cstore a0_r0 rf[z]
intt rf[z] rf[z] q0
simd_mul_mod rf[z] rf[x] rf[y] q0
ntt rf[y] rf[y] q0
ntt rf[x] rf[x] q0
cload rf[y] c0_r0
cload rf[x] b0_r0
NTT Micro-sequencer
CE
CE SIMD Controller
56K Byte Register File
32b Butterfly
32b Butterfly
128 tiles, 64 butterflies per tile

Fig. 2. Accelerator hardware (CE) and software (everything else) stages. The blue region represents the areas formally verified at this time.

sequence of semantic-preserving program transformations on FHE operations and an array of $8k$ butterfly module that carries out 32-bit modular arithmetic operations. These modules are interspersed with an example of an FHE multiplication operation going through the sequence of transformations:

- **FHE-to-Poly Translation** transforms an FHE operation into a sequence of poly-ring operations.
- **RNS Decomposition** transforms a poly-ring operation into a sequence of smaller poly-ring operations based on the Residue Number System (RNS) theory (*e.g.*, 512-bit into 16 x 32-bit coefficients supported natively in hardware).
- **Number Theoretic Transform (NTT)** transforms a poly-ring multiplication operation into a sequence of two NTT operations for the two operands, followed by an element-by-element poly-ring multiplication operation and an inverse NTT (iNTT) operation on the result.
- **SIMD Assembler** transforms poly-ring operations to a sequence of SIMD instructions to load the data from memory to the register file of a CE, carry out the computation, and save the result back from the register file back to the memory. Further optimizations are done on the SIMD instruction stream to minimize computation and data movement.
- **NTT Micro-sequencer** performs an additional transformation to decompose an NTT/iNTT instruction to a sequence of micro NTT/iNTT stage-wise operations.
- **SIMD Controller** transforms SIMD instructions into a vector of parallel element-wise modular-arithmetic instructions to be executed on the $8k$ 32-bit butterfly modules.

With the FHE accelerator execution viewed in this way, the end-to-end proof of correctness can be achieved by proving that each module is semantics preserving and the composition of these modules down to the butterfly units is semantically equivalent to the original FHE operations by transitivity. Among the six modules, the first five modules are implemented through a front-end compiler and only the **SIMD Controller** is implemented in hardware. At this time, we have fully formally verified the CE hardware and the NTT micro-sequencer module that will be presented in the rest of the paper. We also plan to verify the remaining modules in the near future to complete the end-to-end proof covering both hardware and software components.

This view of accelerator design and verification is very similar to compiler passes where each pass applies different code optimizations. Furthermore, each module (pass) can be verified individually and once verified, these modules can be composed in any order (where it makes sense) while maintaining overall correctness.

III. Micro-Architecture Algorithm Verification

For performance reasons, real-world implementations of FHE designs often involve complicated optimizations, *e.g.*, RNS and NTT are employed in modular polynomials multiplications. Optimization tricks such as picking NTT-friendly primes [5] can reduce the computation complexity by a huge factor. Unfortunately, designers often make assumptions and decisions about the optimization process that are often not formally documented. This makes the verification more difficult by introducing gaps between the algorithmic specification and the real implementation. This section proposes a solution to formally capture and verify each of the algorithmic optimization steps using the Dafny program verifier [6].

A. Dafny Program Verifier

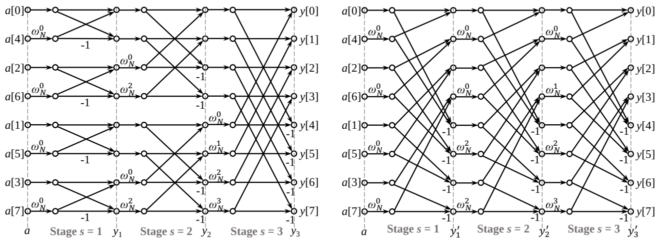
Dafny is a programming language that allows users to write specification, implementation, and proofs of programs. The Dafny static program verifier uses Satisfiability Modulo Theories solver to verify the correctness of the programs.

```
1 function MultiplicativeInv(a: int, n: int): (a’: int)
2   requires 0 < a ∧ 0 < n ∧ coPrime(a, n) ▷precondition
3   ensures a * a’ % n == 1 % n             ▷postcondition
4 {
5   ...// implementation of Multiplicative Inverse omitted
6 }
```

The above code shows an example of the *modular multiplicative inverse* function in Dafny. It takes an integer a and modulus n, and returns an integer, say a', such that $a \cdot a' \equiv 1 \pmod{n}$. Line 2 and line 3 are the pre- and post-condition of the function, which serves as the specification of the function. The pre-condition requires that a and n are positive numbers, and a and n are coprime, only under which the multiplicative inverse exists. Dafny will statically verify that the implementation (function body) indeed satisfies the post-condition for all satisfying inputs. We are using Dafny to model and verify the key algorithms and their optimizations that are implemented in our FHE accelerator design.

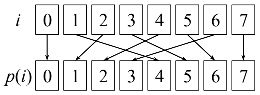
B. NTT Algorithm Verification

Multiplications of large-degree polynomials are pervasive in FHE, therefore efficient algorithms for polynomial multiplication are critical for FHE accelerators. Fast Fourier Transform (FFT) [7] is often used reduce the complexity of polynomial multiplication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. NTT [8] is essentially FFT defined over a finite ring. Mathematically, they are efficient algorithms to compute *discrete Fourier transform* (DFT) of the polynomial coefficient vector. Fig. 3 (a) shows the dataflow chart of the schoolbook NTT algorithm of 8 inputs (a), where array a is in a *bit-reversal permutation* of the indices. The computation is performed in 3 stages. It is easy to see that the routing structure among stages are different, *i.e.*, the indexing pattern to the input and output array is different among stages. Fig. 3 (b) shows the *constant geometry* NTT, with each stage shares the identical routing structure, which makes it more hardware implementation friendly.



(a) (b)

Fig. 3. NTT dataflow chart with 8 inputs: (a) schoolbook 3-loop version [7]. (b) constant geometry 2-loop version.



$$p(i) = \begin{cases} \frac{i}{2}, & i \text{ is even} \\ \frac{i}{2} + \frac{N}{2} & i \text{ is odd} \end{cases}$$

(a) (b)

Fig. 4. (a). The perfect shuffle of an 8-element vector. (b). The mathematical definition of perfect shuffle of index i in a N-element (power of 2) vector.

Although the schoolbook FFT or NTT has been verified in the past, we want to formally verify that the optimized NTT we implement in the hardware is indeed correct. The verification strategy involves 1) verifying that the schoolbook NTT is equivalent to DFT, and 2) verifying that the constant geometry NTT is a refinement of the schoolbook NTT. The intuition behind the algorithm is that we can rearrange the order of the intermediate nodes at each stage (y_1, y_2, and y_3) in Fig. 3 (a), so that the "routing structure" is identical among stages. It turns out that this "rearrangement" algorithm is based on a "perfect shuffle" [9]. Fig. 4 (a) shows an example of the perfect shuffle of an 8-element vector. The shuffling operation[1] $p(i)$ of index i of range $[0, N)$ is shown in Fig. 4 (b), where N is power of 2. Careful examination of the definition shows that this operation is equivalent to the *right rotation* of the binary form of i, *i.e.*, $p(i) = \textsc{RightRotate}(i, N)$. One nice property of bit rotation is that rotating index i (s-bit long) s times leaves i unchanged, *i.e.*, $i = \textsc{Apply}(\textsc{RightRotate}(i, N), \log N)$. With this definition, we can easily reason that constant geometry NTT is equivalent to the schoolbook NTT. In Fig. 3, the invariant for the equivalence reasoning is $y'_s[i] = y_s[i']$, where $i' = \textsc{Apply}(\textsc{RightRotate}(i, 8), s)$, for $i = 0, 1, \ldots, 7$, and $s = 1, 2, 3$. We can see that $y_3 = y'_3$ (indices are 3-bit long).

[1] As the input to NTT are in a *bit-reversal permutation* of the original input vector, this is the inverse of the definition of *perfect shuffle* published in [9].

C. Modular Multiplication Algorithm Verification

At the high level, NTT algorithm helps us bring the complexity of polynomial multiplication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, where n is the degree of the polynomial. At low level, we still need to multiply numbers in the form of modular multiplication, *i.e.*, in the form of $a \cdot b \pmod q$. Efficient algorithms have been invented in the the past, *e.g.*, Montgomery reduction [10] algorithm. This algorithm requires the inputs a and b to be in Montgomery form, *i.e.*, $aR \pmod q$ and $bR \pmod q$, where the auxiliary modulus R is coprime to q. As a result, when we perform modular multiplication of the two inputs, we get $(abR)R \pmod q$. Thus, a *reduction* step is needed to *remove* the extra factor of R. Listing 1 shows the multi-word Montgomery reduction algorithm [10]. This allows us to use smaller multipliers to reduce long-bitwidth inputs.

Listing 1. Montgomery Reduction algorithm: Given number T to be reduced that has n words of base B, modulus q and q', auxiliary modulus R and R', it outputs S such that $T \cdot R' \equiv S \pmod q \wedge S < q$

```
 1 method REDC(q: nat, n: nat, R: nat, R': nat, q': nat,
 2             T: nat) returns (S: nat)
 3   requires n > 0 ∧ q > 0 ∧ q' > 0 ∧ R' > 0 ∧ T > 0
 4   requires 0 < q < Pow(B(), n)      ▷ $0 < q < B^n$
 5   requires R == Pow(B(), n)         ▷ $R = B^n$
 6   requires R * R' % q == 1 % q      ▷ $R \cdot R^{-1} \equiv 1 \pmod q$
 7   requires coPrime(q, B()) ∧ coPrime(R, q)
 8   requires q * q' % B() == -1 % B() ▷ $q \cdot q' \equiv -1 \pmod B$
 9   requires T < q * R
10   ensures S < q
11   ensures S % q == T * R' % q       ▷ $T \cdot R^{-1} \equiv S \pmod q$
12 {
13   var a: nat := T;
14   for i: nat := 0 to n {   ▷ reduce 1 word per iteration
15     var ui: uint := a % B() * q' % B();
16     a := a + ui * q;
17     a := a / B();
18   }
19   S := if a ≥ q then a - q else a;
20 }
```

Note that the listing only shows the specification and implementation of this algorithm, The actual proof is omitted. Line 3 to 9 are the pre-conditions of function `REDC`, and line 10 to 11 are the post-conditions that we need to verify.

Proof. a on Line 13 is represented as a list of words of base B, $a = (a_{2n-1}, \cdots, a_1, a_0)_B$, the loop body is equivalent to

```
for i: nat := 0 to n {
  var ui: uint := a[i] * q' % B();
  a := a + ui * q * Pow(B(), i);
}
a := a / R;            ▷ $R = B^n$
```

After the loop iterations, a is divisible by R. For every loop iteration, a is incremented by $u_i q B^i$, so at the end of the loop, $a = T + q \sum_0^{n-1} u_i B^i$. Let $a'R = a$, then $a'R \equiv (T + q \sum_0^{n-1} u_i B^i) \pmod q$. Therefore $a'R \equiv T \pmod q$, and $a' \equiv TR^{-1} \pmod q$.

Now, we need to prove that a' is in the range from 0 to $2q$, so that after then range correction on line 19, the final output S is in the range from 0 to q. We know $T < qB^n$, and $\sum_0^{n-1} u_i B^i < B^n$ (*e.g.*, $999 < 10^3$ for $n = 3$ and $B = 10$), so $a/B^n < 2q$ after the loop iterations, and $S < q$. □

Notice that we can choose arbitrary modulus q in Listing 1 as long as it satisfies the pre-conditions. If we smartly choose q to be the form [5] of $q = kB + 1$ for some k, we can further simplify the loop body. Listing 2 shows the optimized Montgomery reduction algorithm with hardware friendly modulus.

Listing 2. Montgomery Reduction algorithm with hardware friendly modulus: $\exists\ k$, such that $q = kB + 1$

```
method REDC'(q: nat, n: nat, R: nat, R': nat, q': nat,
             T: nat) returns (S: nat)
  requires ∃ k: nat · q == k * B() + 1 ▷ hw friendly q
  ... // omitting same pre/post-conditions as in REDC
{
  ghost var k: nat :| k * B() + 1 == q;
  var a: nat := T;
  for i: nat := 0 to n {
    var ui: uint := a % B() * q' % B();
    ghost var ah, a0 := a / B(), a % B();
    ghost var carry := if a0 == 0 then 0 else 1;
    a := ah + TwosComplement(a0) * k + carry;
  }
  S := if a ≥ q then a - q else a;
}
```

Because $q = kB + 1$, then $(kB + 1)q' \equiv -1 \pmod{B}$, which further implies $q' \equiv -1 \pmod{B}$. Therefore, in the loop body, $u_i = a_0 \cdot q' \pmod{B}$ (a_0 is lowest word in a) can be simplified into $u_i = -a_0 \pmod{B}$. Further, since $0 \leq a_0 < B$, $-a_0 \pmod{B} = B - a_0$ *if* $a_0 \neq 0$ *else* 0. This is essentially the *two's complement* of a_0: $u_i = \text{TwosComplement}(a_0)$. By partitioning a into two parts (higher words and lowest word: $a = a_h B + a_0$), $a \leftarrow a + u_i q$ is rewritten to $a \leftarrow a + u_i(kB+1)$. After line 17 of Listing 1, $a \leftarrow (a_h + u_i k) + (a_0 + u_i)/B$. Since $u_i = B - a_0$ *if* $a_0 \neq 0$ *else* 0, $(a_0 + u_i)/B0 = 1$ *if* $a_0 \neq 0$ *else* 0. Then $a \leftarrow a_h + \text{TwosComplement}(a_0) \cdot k + carry$, where $carry = 1$ *if* $a_0 \neq 0$ *else* 0.

IV. Hardware Implementation Verification

Having verified the functionality of the core algorithms, verifying the RTL implementation boils down to applying formal equivalence verification between the RTL and the algorithms. Due to stringent performance and area requirements, the RTL is manually designed which further exemplifies the need for verification. Despite having refined the algorithms to be "hardware optimized" and the RTL having a close structural similarity to the algorithms, RTL equivalence verification was still not straight-forward for several reasons. Firstly, and not surprisingly, is due to complexity and scalability management. Secondly, the RTL has lower-level implementation details that impact design latency, for example, that is not reflected in the algorithm. Lastly, specific to NTT micro-sequencer verification, the RTL only implements "one stage" of the NTT algorithm and the full NTT functionality is realized through a sequence of "one-stage instructions" in software. Consequently, the full end-to-end NTT verification needs to reason about both the hardware and software sequence.

We use Forte [11], a formal verification environment used heavily for CPU and floating-point hardware verification, for this work. We leverage Symbolic Trajectory Evaluation [12] built into Forte for the primary RTL verification procedure and the FL programming language to codify the "RTL specification" and the hardware and software "contract" for NTT micro-sequencer verification. The specification codified in FL is the bridge between the algorithms verified in Dafny and the hardware+software implementation.

A. Butterfly Unit Verification

The butterfly unit is at the core of all polynomial operations in the accelerator where the actual modular arithmetic computations are done. To verify this unit, an RTL specification is defined in FL that captures the behaviors expected for different butterfly operations (*e.g.*, add, sub, mul, mac, etc.) including the Modular multiplication algorithm verified in section III-C. Fig. 5 illustrates an abstract view of the butterfly RTL showing the front-end which does input processing, the middle section that does the multiplication and Montgomery reduction, and the back-end that handles addition/subtraction operations and final modulo adjustments. To mitigate complexity blow-up, the verification was divided into six parts. This partitioning was primarily dictated by the need to black-box the multipliers (boxes with "*") as these are well-known to cause computational blow-up. Since standard library multipliers are used in the design that have been previously proven, we partitioned the design around the multipliers and applied a divide-and-conquer approach with assume-guarantee checks on the multiplier input/output interfaces.

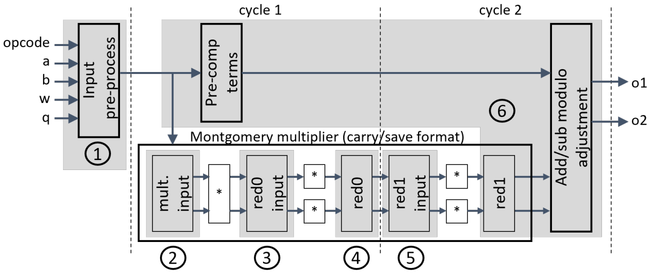


Fig. 5. Butterfly RTL abstract view and verification partitioning.

To avoid having to model the low-level details at the algorithm level, the bridge specification in FL modeled the correspondence between the RTL level information and the algorithm. For example, the multiplication and Montgomery reduction unit was implemented in carry-save format in RTL. In this case, details of how the carry and save vectors are implemented and handled through the entire unit is irrelevant to the algorithm as long as the sum of the *carry* and *save* vectors matches what the algorithm expects. Similarly, the final modulo adjustment logic speculatively computes different values in parallel and picks the right value for the output. From the verification point-of-view, the details of how the speculation is done is irrelevant as long as the final value matches the algorithm. By abstracting the verification target in terms of the sum (for the carry-save implementation) or the final output (for the speculation logic), the verification flow becomes agnostic to these low-level details. Indeed, the verification partitioning/flow did not need any update when the speculation logic in partition 6 was modified for timing reasons.

B. NTT Verification

Verification of the NTT functionality had different challenges due to 1) model size issues, and 2) verification of the full NTT algorithm requiring co-verification of the hardware and micro-sequencer generated instructions. The model size issue is due to NTT operations requiring inter-tile data exchanges and consequently having to reason about the entire compute engine with all the tiles. The co-verification challenge, on the other hand, requires that both hardware and micro-sequencer be modeled and reasoned about in a unified way. Note that the arithmetic operations involved in the NTT operation was verified as part of the butterfly unit. As such, the butterfly units were black-boxed and it is sufficient to verify 1) that the correct data is provided to the butterfly unit inputs, and 2) the outputs passed from one stage to the next according to the NTT algorithm.

We use a hierarchical approach to capture the behavior of the RTL at different levels of hierarchy and a compositional proof to verify the entire logic. Recall that the CE is built from an array of tiles. To implement the inter-tile data exchange required by the constant geometry NTT algorithm, data is reordered at different sections of the design starting from inside of a "producer" tile, to the tile outputs, all the way to the CE level, and similarly when data is routed back as inputs to the "consumer" tile.

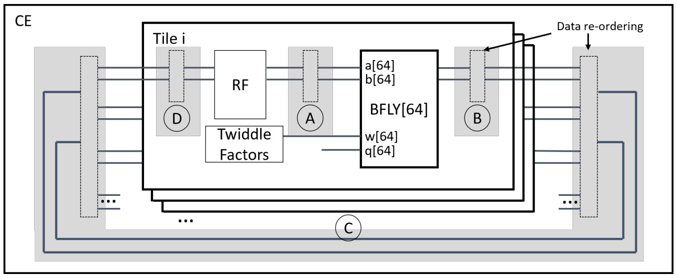
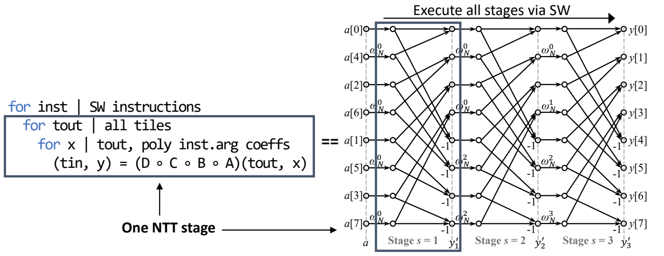


Fig. 6. Compute Engine hierarchical routing for NTT.

Fig. 6 shows an abstract view of the RTL hierarchy with tiles instantiated. The full NTT data-path includes the RF (register files) that holds the input/output data, the butterfly units, and the reordering logic (shown as vertical bars) that implements the inter-tile data exchange. To create a scalable proof, specifications for each of the reordering logic was first captured in FL and verified against the RTL. In the abstract view above, the reordering specification for A, B, and D was verified with a single tile instance. Similarly, the reordering specification for C was verified using a model of the CE but with all the tiles black-boxed. At no time was the entire CE RTL, with all the tiles, required to be built.

Having verified the different reordering specifications, they are then composed to model the entire routing network. The verification then ensures that for all coefficients, the composed routing specification matches the algorithm. Specifically, the composition of $(D \circ C \circ B \circ A)$ is a mapping from (producer tile, coefficient x) to (consumer tile, coefficient y), where x and y are coefficient indices within the respective tiles, according to the algorithm. Fig. 7 illustrates this mapping where the boxed-in regions represent one stage of the NTT algorithm that is implemented in hardware. As the RTL implements the NTT operation over a static number of tiles and coefficients, it then suffices to check that the composition over all tiles and coefficients corresponds to the behavior of one stage of the constant-geometry NTT algorithm.



```
for inst | SW instructions
  for tout | all tiles
    for x | tout, poly inst.arg coeffs
      (tin, y) = (D ∘ C ∘ B ∘ A)(tout, x)
```

Fig. 7. Specification composition and mapping to NTT algorithm. Boxed-in region represents the mapping for a single NTT stage.

The same framework was then used to extend the proof to include the NTT micro-sequencer. By composing the software instruction sequence "contract" over the "one-stage proof", the full NTT functionality over multiple stages was verified. The proof effectively verified that the NTT micro-sequencer composed with the CE "one-stage" hardware (blue region in Fig. 2), matches the NTT algorithm.

The RTL also implements the computation of the twiddle factors used in the NTT instructions. These twiddle factor updates and usage were verified as part of the NTT verification based on the NTT algorithm as well. INTT verification is very similar to NTT with the main differences in the routing and twiddle update specifications.

C. Other logic

While the butterfly unit and NTT functionality were the primary targets for formal verification due to their highly optimized implementations, verifying all other logic was equally critical to ensure overall correctness of the hardware. This includes verifying instruction and data loads into the accelerator, RF reads/writes, instruction decode/execution, correct butterfly unit opcode and data according to the instruction, and that butterfly unit outputs are sent to the right place.

V. Results

In this work, we formulated a full end-to-end verification strategy covering both hardware and software components of the FHE accelerator. At this time, we have completed the verification of the CE hardware component and NTT micro-sequencer module. The verification of these components starts from the mathematical definition of the key algorithms, through various "hardware friendly" algorithmic refinements, and finally to the actual RTL implementation with low-level design details. Verifying the mathematical definition and equivalence of these key algorithms, despite being well-known, is critical to establish a baseline proof that incremental refinements can be compared against. While we did not find issues in the algorithms (as expected, but even better verified), the other big benefit of having a hardware optimized algorithm is that the RTL was structurally similar to it, making the verification task more manageable. It would be a lot more difficult to verify the final optimized RTL implementation against the high-level algorithms due to the the bigger abstraction gap.

TABLE I
BUG COUNT BY RTL UNIT

RTL Unit	Butterfly	Tile	CE
Bug count	5	12	4

Overall, we found 21 issues in the RTL as shown in Table I. Bugs found range from simple (*i.e.*, would be found with a reasonable set of test vectors) to ones that would be hard to detect with random stimulus. For example, we found a bug in the butterfly logic shown in Fig. 8 where due to the misinterpretation of the top bits, the output mux selected the wrong speculative result. This only happens when the input values for w, b, and q are such that partial product multiplier generated specific values in bits [33:32] of the carry/save outputs. This issue was detected by the formal verification flow despite having running a million random test vectors in simulation.

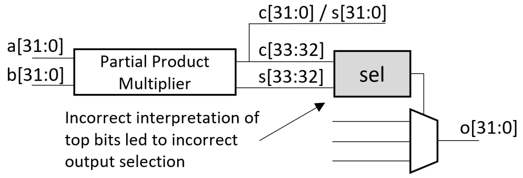

c[31:0] / s[31:0]
a[31:0]
b[31:0]
Partial Product Multiplier
c[33:32]
s[33:32]
sel
Incorrect interpretation of top bits led to incorrect output selection
o[31:0]

Fig. 8. Butterfly unit output selection bug.

In the tile RTL, we found a bug where loading instructions into the tile also caused data in the RF to be modified due to the RF write enable signal getting set incorrectly on an instruction register write in some cases. As the RF write address bus could have any value at that time, random RF locations would get updated making this bug hard to detect or reproduced with random simulation. In general, these type of "non-interference" issues would need explicit checks in the model to be caught immediately. With formal verification, we found this issue by detecting unexpected write conditions during the RF verification. In the CE RTL, we found a bug in the twiddle factors update logic for polynomials of size $\geq 32k$. The verification flow detected the discrepancy resulting from the incorrect twiddle factor index getting updated while evaluating higher numbered stages generated by the NTT micro-sequencer for these larger polynomials.

An additional benefit of doing the formal verification work is the detection of unnecessary logic activity. Specifically, we found conditions when the RF is read but the resulting data is not used. Avoiding these RF reads helped improve power consumption without impacting functionality.

The full algorithm and RTL equivalence verification completes in one hour using about 10GB of memory. By using a composition-based divide-and-conquer strategy, we built a scalable verification flow that can be extended as needed (*e.g.*, to support proofs for bigger polynomials, more tiles, etc.). All proofs are codified and used for both verification and regression checks (depending on the complexity of changes). We were able to provide immediate feedback that a proposed bug fix indeed resolved an issue but also broke previously working logic, for example. With the butterfly unit being quite mature at this point, we expect at most some local tweaks (*e.g.*, for timing) that should not impact the verification partitioning and flow. For NTT, different data exchange routing configuration continue to be investigated and due to the way we captured the various reordering steps, all that's needed is to update the reordering specification and re-use the same composition proof to verify the new design as before.

VI. CONCLUSION

In this paper, we presented a scalable approach for formally verifying the correctness of the compute engine against its ISA specification in an FHE accelerator, based on the composability of program transformations well established in compiler verification. To our best knowledge, this is the first time such an idea is applied to hardware verification to make it scalable. For the future work, we plan to auto-generate the behavioral specification for RTL verification in Forte from the algorithmic specification in Dafny. We also plan to extend the scope of verification to cover the entire compiler stack, and to other programmable accelerators.

REFERENCES

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford university, 2009.

[2] "DARPA Selects Researchers to Accelerate Use of Fully Homomorphic Encryption," https://www.darpa.mil/news-events/2021-03-08, Mar 2021.

[3] E. Singh, F. Lonsing, S. Chattopadhyay, M. Strange, P. Wei, X. Zhang, Y. Zhou, D. Chen, J. Cong, P. Raina, Z. Zhang, C. Barrett, and S. Mitra, "A-qed verification of hardware accelerators," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[4] J. A. Navas, B. Dutertre, and I. A. Mason, "Verification of an optimized ntt algorithm," in *Software Verification*, M. Christakis, N. Polikarpova, P. S. Duggirala, and P. Schrammel, Eds., 2020.

[5] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *22nd Euromicro Conference on Digital System Design*, 2019, pp. 253–260.

[6] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[8] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, "On the design of hardware building blocks for modern lattice-based encryption schemes," ser. CHES'12, Berlin, Heidelberg, 2012, p. 512–529.

[9] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. 20, no. 2, p. 153–161, feb 1971.

[10] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[11] R. Jones, J. O'Leary, C. Seger, M. Aagaard, and T. Melham, "Practical formal verification in microprocessor design," *IEEE Design Test of Computers*, vol. 18, no. 4, pp. 16–25, 2001.

[12] C. Seger and R. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories." *Formal Methods in System Design*, vol. 6, no. 2, p. 147–189, 1995.