Towards A Correct-by-Construction FHE Model

Zhenkun Yang, Wen Wang, Jeremy Casas, Pasquale Cocchini, Jin Yang
Intel Labs, Intel Corporation

***Abstract*—This paper presents a correct-by-construction method of designing an FHE model based on the automated program verifier Dafny. We model FHE operations from the ground up, including fundamentals like GCD, coprimality, Montgomery multiplications, and polynomial operations, etc., and higher-level optimizations such as Residue Number System (RNS) and Number Theoretic Transform (NTT). The fully formally verified FHE model serves as a reference design for both software stack development and hardware design, and verification efforts. Open-sourcing our FHE Dafny model with modular arithmetic libraries to GitHub is in progress.**

I. Introduction

Fully Homomorphic Encryption, in short FHE, comprises cryptosystems that supports arbitrary computation on an encrypted data set. Efficient implementations of FHE programs involve transformations between various data representations and complex arithmetic defined on large data fields. Together with the complexity brought by algorithm-level and architectural-level optimizations, the difficulty of mapping FHE programs to hardware or software systems while maintaining functional correctness constitutes a rather challenging task.

Formal verification is a powerful technique often used for mathematically proving the correctness of intended functionality in hardware or software systems with respect to their formal specification. In comparison to validation, which is a stimulus-oriented approach to ensure that the system under test works on a set of pre-defined inputs, formal verification systematically proves that the implementation conforms to the specification.

A. FHE

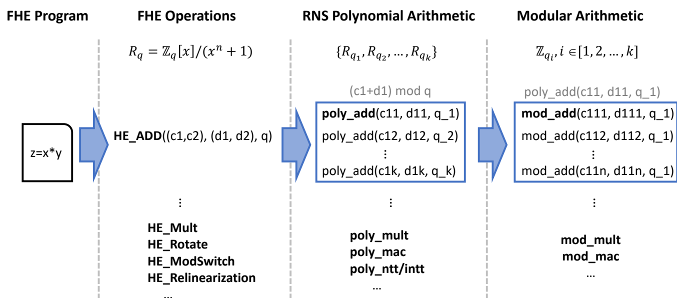

Fig. 1. FHE decomposition

In our work, we focus on the modelling and verification of an FHE program which performs homomorphic computation for a given function. Figure 1 shows an example of the decomposition flow for an FHE program. An FHE program is composed of a sequence of homomorphic operations, e.g., homomorphic multiplication, homomorphic rotation, etc. The inputs for the homomorphic computation are ciphertexts composed of arrays of polynomials in a polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. The coefficient modulus q can be chosen to be composed of several distinct prime values, i.e., $q = q_1 \times q_2 \cdots \times q_k$. In this case, the polynomials in the ciphertext can take advantage of the Residue Number System (RNS) representation which splits a polynomial in R_q into k polynomials defined in $\{R_{q_1}, R_{q_2}, \cdots, R_{q_k}\}$, where $R_{q_i} = \mathbb{Z}_{q_i}[x]/(x^n+1)$ for $i = 1, \cdots, k$. Here, RNS enables an efficient decomposition of FHE operations to polynomial operations that are defined with smaller moduli (or RNS residues), i.e., $q_1, q_2, \cdots, q_k$. These RNS polynomial operations, e.g., polynomial addition and polynomial multiplication, can further be realized through the underlying modular arithmetic defined on the field $\mathbb{Z}_{q_i}$, for $i = 1, \cdots, k$.

For performance reasons, real-world implementations of FHE often involve complicated optimizations, e.g. RNS and Number Theoretic Transform (NTT) as employed in modular polynomials multiplications. Optimization tricks such as picking NTT-friendly primes [1] can reduce the computation requirement by a huge factor. As a consequence, however, designers often make specific assumptions about the optimization process (often not formally documented) from the original design. This makes the verification harder by introducing gaps between the algorithmic specification and the real implementations. This paper proposes an approach to formally capture and verify each of the optimization steps in a formal language.

This paper aims at presenting a correct-by-construction FHE model with formal verification techniques. More specifically, we use the Dafny [2] program verifier to formally model and verify all the operations of an FHE system. We write both specification and implementation in Dafny, a high-level language designed with verification support. Dafny uses the Z3 [3] SMT solver for proof automation.

B. Dafny

```
1  function modinv(a: int, n: int): int
2    requires n > 0                          ▷precondition
3    ensures a * modinv(a, n) % n == 1 % n  ▷postcondition
4  {
5    // implementation of modular inverse omitted
6  }
```

The above code shows an example of the *modular multiplicative inverse* function `modinv` in Dafny. It takes an integer `a` and modulus `n`, and returns an integer, say `v`, such that `a` $\times$ `v` $\equiv$ `1(mod n)`. Line 2 and line 3 are the pre- and

post-condition of the function, which serves as the specification of the function. Dafny will statically verify that the implementation (function body) indeed correctly implements the specification for all possible inputs. Similarly, the post-condition for modular polynomial multiplication can be easily modeled regardless of complicated optimizations.

II. Correct-by-construction approach

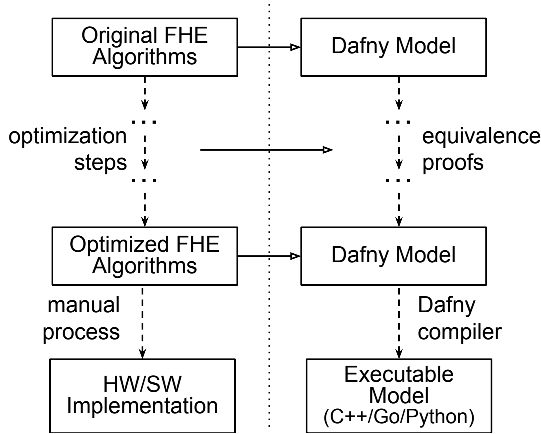

Fig. 2. Design and verification flows of an FHE model

Figure 2 shows the overall design and verification work flows. The left-hand side details a conventional design flow where designers perform a number of algorithm optimization steps starting from a set of original FHE algorithms. Most often this is done in an informal way, with limited simulation based testing that is focused only to specific scenarios. Indeed, simulation based validation approaches are predominantly used to test that the end design works on a limited set of pre-selected inputs. However, simulation based testing can only show the presence of bugs, not their absence. Since these optimization steps are not formally verified, bugs or incorrect assumptions may be unwillingly introduced. Additionally, during subsequent verification stages, such a workflow creates a potentially big gap between the original algorithms and their actual implementations, therefore making any later formal correctness reasoning of the implementations more difficult.

On the right-hand side, we propose a new end-to-end correct-by-construction design and verification flow. We start modeling the algorithms from the very beginning. Each and every optimization step is modeled to ensure correctness along the way. This not only makes the verification task easier by taking advantage of smaller optimization steps, but also enables the identification of issues as early as possible with more accurate root causing.

At the end of the flow, executable models written in different target languages (C++, Go, etc.) are generated with the Dafny compiler. This provides an executable FHE model to higher-level software stack components, such as FHE compiler and FHE workload. It also serves as a reference model for downstream hardware designs. It is worth noting that the proof strategies and pre- and post-conditions specified in the FHE Dafny models have the added value of providing guidance for the formal verification effort in the hardware design and validation phases. Beyond these benefits, the generated models can be further synthesized into RTL designs with C/C++ based High-Level synthesis tools or emerging domain-specific languages such as HeteroCL [4] and PyLog [5].

A. FHE model construction strategy

The construction of an FHE model starts from the lowest level of the arithmetic, i.e., the modular arithmetic. Apart from standard modular addition and modular multiplication arithmetic, we also model arithmetic that are specifically utilized in a given FHE program, such as the modular multiply-and-accumulate arithmetic. In the Appendix IV, we show an example of the modeling and verification of modular multiplication arithmetic, where the Montgomery reduction algorithm [6] is applied with an optimization utilizing NTT-friendly primes [1]. After formally verifying and modeling the modular arithmetic, these low-level models can be treated as a black-box and be used further as underlying arithmetic for higher-level arithmetic. For example, the RNS polynomial arithmetic can be constructed based on the modular arithmetic; similarly, we can construct FHE operations by use of the RNS polynomial arithmetic. With all the FHE operations modeled and formally verified, we can construct a complete FHE model which is formally verified once the model is constructed.

III. Conclusion

This paper presents a correct-by-construction approach for the modeling and formal verification of FHE models. We leverage the SMT-based automated theorem prover Dafny to formally verify high-level algorithms and optimizations steps. We generate fully verified models in selected target languages, e.g. C++, Go, for integration with the rest of the software system, i.e. compiler and workloads. This also provides a reference model and formal verification guidance for hardware designs. We do plan to open-source our effort in the near future.

IV. Appendix: Montgomery Reduction in Dafny

This section shows an example of the Montgomery Reduction [6] algorithm modeled and verified in Dafny. The proof is lengthy, this is due to the fact that non-linear integer arithmetic is in general undecidable, thus unstable in Z3, so we turned off non-linear arithmetic heuristics in Danfy, and call non-linear arithmetic lemmas from a library [1] explicitly.

```
1  // R: auxiliary modulus, R' is modular inverse
2  // N: the modulus, NN' = -1 (mod R)
3  // T: Input to be reduced
4  method REDC(R: nat, R': nat, N: nat, N': nat, T: nat)
5    returns (S: nat) // Output after the reduction
6    requires 0 < R && 0 < R' && 0 < N && 0 < N'
7    requires IsCoPrime(R, N)
8    requires R * R' % N == 1 % N
9    requires N * N' % R == -1 % R
10   requires N < R && T < N * R
11   ensures S % N == T * R' % N && S < N
12 {
```

[1] https://github.com/dafny-lang/libraries

```
13    var m: nat := T % R * N' % R;
14
15    LemmaMulNonnegative(m, N); // m * N >= 0
16    LemmaDivPosIsPos(T + m * N, R);  // prove t >= 0
17
18    var t: nat := (T + m * N) / R;
19
20    // prove T + m * N is divisible by R
21    assert (T + m * N) % R == 0 by {
22      LemmaREDCDiv(T, R, N, N');
23    }
24
25    // range correction
26    S := if t >= N then t - N else t;
27
28    // prove the range of t
29    assert t < 2 * N by {
30      assert N * m < N * R by {
31        LemmaMulLeftInequalityAuto();
32      }
33      assert T + N * m < (N * R + N * R);
34      assert T + N * m < (N + N) * R by {
35        LemmaMulIsDistributiveAddOtherWayAuto();
36      }
37      assert T + N * m < (2 * N) * R;
38      assert T + N * m < R * (2 * N);
39      assert (T + N * m) / R < 2 * N by {
40        LemmaMultiplyDivideLtAuto();
41      }
42    }
43    // prove the range of return S
44    assert S < N;
45
46    assert T+m*N == R * ((T+m*N) / R) + (T+m*N) % R by {
47      LemmaFundamentalDivModAuto();
48    }
49    assert T + m * N == R * ((T + m * N) / R);
50    assert T + m * N == R * t == t * R;
51
52    // final property
53    assert t % N == T * R' % N by {
54      LinearCongruenceAndMultiplesVanish(t, R, R', T, m, N);
55    }
56
57    // prove S is congruent to t mod N
58    assert (t - N) % N == t % N by {
59      LemmaModSubMultiplesVanishAuto();
60    }
61    assert S % N == t % N;
62  }
63
64  /* let m = T % R * N' % R, and N * N' % R == -1 % R
65     then, T + m * N is divisible by R
66     idea: add multiple (m) of N to T, T divisible by R */
67  lemma LemmaREDCDiv(T: nat, R: nat, N: nat, N': nat)
68    requires 0 < R && 0 < N && 0 < N'
69    requires N * N' % R == -1 % R
70    ensures (T + (T % R * N' % R) * N) % R == 0
71  {
72    var m := T % R * N' % R;
73    var r := T % R;
74    // prove T + m * N is divisible by R
75    assert (T + m * N) % R == 0 by {
76      calc == {
77        (T + m * N) % R;
78          { LemmaAddModNoopAuto(); }
79        (T % R + m * N % R) % R;
80        (T % R + (T % R * N' % R) * N % R) % R;
81        (T % R + (r * N' % R) * N % R) % R;
82          { LemmaMulModNoopGeneralAuto(); }
83        (T % R + (r * N' * N % R)) % R;
84          { LemmaMulIsAssociativeAuto();
85            LemmaMulIsCommutativeAuto(); }
86        (T % R + (N * N' * r % R)) % R;
87          { LemmaMulModNoopGeneralAuto(); }
88        (T % R + (N * N' % R * r % R)) % R;
89        (T % R + (-1 % R * r % R)) % R;
90          { LemmaMulModNoopGeneralAuto(); }
91        (r + -1 * r % R) % R;
92        (r + -r % R) % R;
93          { LemmaAddModNoopAuto();
94            LemmaModBasicsAuto(); }
95        (r % R + -r % R) % R;
96          { LemmaAddModNoopAuto(); }
97        (r + -r) % R;
98        0 % R;
99          { LemmaModEquivalenceAuto(); }
100       0;
101     }
102   }
103 }
104
105 /* let xR = b + kN, where R and N are coprime
106    then x = R'b (mod N), R' is the modular inverse of R */
107 lemma LinearCongruenceAndMultiplesVanish(
108   x: int, R: int, R': int, b: int, k: int, N: int)
109   requires R > 0 && N > 0
110   requires R * R' % N == 1 % N
111   requires IsCoPrime(R, N)
112   requires x * R == b + k * N
113   ensures x % N == R' * b % N
114 {
115   calc == {
116     (b + k * N) % N;
117       { LemmaMulIsCommutativeAuto(); }
118     (N * k + b) % N;
119       { LemmaModMultiplesVanishAuto(); }
120     b % N;
121   }
122   calc == {
123     x % N;
124       { LinearCongruence(R, R', x, N, b); }
125     b * R' % N;
126   }
127 }
```

References

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

[2] 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.

[3] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[4] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[5] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-m. Hwu, "Pylog: An algorithm-centric python-based fpga programming and synthesis flow," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 227–228.

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