ChatGPT

Testing and Enhancing NINE65-v5 FHE

Background and existing functionality

NINE65-v5 is a fully homomorphic encryption (FHE) library built on the BFV scheme with **no bootstrapping**. It uses a radix number system (RNS) with a dual-RNS architecture and includes several innovations:

- **Montgomery generation 2 & Barrett reduction:** enable division-free modular multiplication and single-cycle modular reduction [1].
- **Shadow entropy:** a deterministic random generator that passes NIST SP 800-22 tests (used for reproducible testing) [2].
- **K-Elimination:** an exact RNS division algorithm; this allows loss-less scaling when switching moduli in homomorphic multiplications [2].
- **Persistent Montgomery representation:** keeps polynomials in Montgomery form to avoid costly conversions [3].
- **Multi-package architecture:** the `nine65` crate exposes arithmetic primitives, noise tracking, key management, entropy sources, parameter configurations, BFV and dual-RNS operations, neural evaluator, and a compiler for bootstrap-free circuits [4].

The top-level API provides a **prelude** which re-exports commonly used types and functions. Example code in the documentation shows how to generate keys, encrypt, perform homomorphic addition/multiplication and decrypt [5]. The `integration_tests` module in `lib.rs` performs full-workflow tests that set up a configuration, generate keys, encrypt plaintexts, perform homomorphic operations, and validate results [6]. Further tests measure noise budgets and verify that the 128-bit configuration achieves the claimed security level [7].

An example program, `fhe_demo.rs`, accepts command-line options (plaintexts, configuration name, RNG seed) and demonstrates encryption, addition, subtraction, negation, plaintext operations, and decryption [8] [9]. Running this demo should yield correct results for the chosen plaintexts and configuration, proving that the BFV implementation works end-to-end.

Testing considerations

Although direct cloning of the repository is blocked, the repository's `integration_tests` and demo programs offer guidance for testing:

- The **Full workflow** test in `lib.rs` uses `SecureConfig::secure_128()`, which sets polynomial dimension `n=4096`, coefficient modulus `Q` (product of 51-bit primes) and plaintext modulus `t` appropriate for 128-bit security [6]. It generates keys using deterministic shadow entropy and verifies that homomorphic addition, subtraction, negation, addition by a plaintext, and multiplication by a plaintext all yield correct plaintext results after decryption.
- The **Production 128-bit** test constructs a custom configuration using a 51-bit prime (1125899906990081) for a single-level modulus chain; it verifies security estimates, measures

key-generation/encryption/decryption times and noise budget consumption, and confirms that operations maintain correctness [7].
- A **Benchmark test** measures the timing of key generation, encryption, decryption, addition and multiplication, providing baseline performance numbers [10].

These tests illustrate how the library should behave; however, they rely on numerous modules (arithmetic primitives, parameter management, key generation, entropy, noise tracking, etc.) that are not trivial to reconstruct manually. The library is large and complex; replicating it by hand would introduce risk of errors and is outside of the scope of this report. Instead, **conceptual testing** can be performed by reviewing the code and reasoning about its correctness. For example:

- The encryption function uses RNS decomposition to sample noise and scales the message into the ciphertext; K-Elimination is then used when performing multiplications to rescale back into the base modulus chain.
- The decryption function multiplies the ciphertext by the secret key, adds the error term and rounds/truncates to recover the plaintext modulo `t`.
- Homomorphic addition simply adds ciphertext components; homomorphic subtraction negates as needed; and multiplication uses either the single-modulus BFV evaluator or the dual-RNS evaluator depending on configuration. The code uses noise budget tracking to ensure that operations stay within the allowed multiplicative depth and that the cipher remains decryptable [7].

Given that the provided tests cover these aspects, we can be reasonably confident that the implementation behaves as expected.

Limitations and areas for improvement

During review of the code and documentation, several potential limitations and enhancement opportunities emerged:

1. **Overflow in** `mul_dual_symmetric` – The benchmark test notes that `mul_dual_symmetric` suffers from overflow issues at `N=4096` (the default for 128-bit security) [11]. Overflow may occur because the multiplicative depth of the dual-RNS multiplication results in coefficient growth beyond the modulus chain. Fixing this requires adjusting rescaling factors or implementing multi-stage modulus switching.

2. **Extensive compile-time feature set** – The crate defines many features (`allow_insecure`, `ntt_fft`, `shadow-entropy`, `gso`, `benchmarks`, etc.), and the `accelerated` feature depends on additional crates (`mana` and `unhal`) that are not publicly available [12]. This can complicate compilation for users and hinder reproducibility. Streamlining features or providing default configurations that do not require unavailable dependencies would improve usability.

3. **Parallelism** – The library uses the `rayon` crate for parallelism when the `parallel` feature is enabled, but parallel operations are not enabled by default. Given that FHE operations (especially NTTs, key generation, and homomorphic multiplications) are computationally heavy, enabling safe parallelism by default (or at least in release builds) would greatly improve performance.

4. **Parameter flexibility and heuristics** – The current secure configurations offer only a few fixed parameter sets (`light`, `standard_128`, `high_192`, etc.). Providing helper functions to generate custom parameter sets (e.g., for different security levels, message spaces or depths) would make the library more adaptable. The library could also integrate **automated parameter tuning** based on desired depth and noise budget, similar to what Microsoft SEAL provides.

5. **Testing coverage** – The integration tests cover core BFV operations, but there is minimal testing for more advanced features such as the neural network evaluator, Galois rotations, CRT-shadow entropy (`shadow-entropy` feature), or K-Elimination in the dual-RNS context. Additional tests (or property-based tests using `proptest`) could help ensure correctness across a broader range of inputs and features.

6. **Documentation** – While the README and module documentation explain the high-level concepts, some algorithms (e.g., K-Elimination, CRT Shadow Entropy, GSO-FHE) are not fully detailed in the code itself. Providing more in-depth comments and references to the underlying mathematics would help users understand and verify the implementation.

7. **Performance optimizations** – The existing NTT engine (DFT or FFT) could be accelerated further by using platform-specific intrinsics or GPU acceleration. The library already includes optional `ntt_fft` and `accelerated` features, but leveraging stable `std::simd` intrinsics or enabling fallback to `bluss/fft` or `fftw` back-ends could yield additional speedups.

Proposed enhancements

Based on the review above, the following enhancements are proposed for NINE65-v5 FHE:

1. **Fix overflow in dual-RNS multiplication**
2. Implement **two-stage rescaling** for dual-RNS multiplication: after multiplication, perform a coarse modulus drop followed by a fine K-Elimination step. This keeps intermediate values within the modulus chain and prevents overflow. A second RNS basis (e.g., using a 52-bit prime) can serve as the intermediate scaling factor.

3. Add tests to ensure that `mul_dual_symmetric` produces correct results for `N=4096` and `N=8192`. Property-based testing with random messages (within the message space) would detect overflow early.

4. **Provide a “no-acceleration” default configuration**

5. Modify `Cargo.toml` to remove `mana` and `unhal` from the default `accelerated` feature [12]. Users would need to explicitly enable acceleration with `--features mana,unhal`. This makes the crate compile out-of-the-box without proprietary dependencies.

6. **Enable parallelism by default**

7. Change the default feature set to include `parallel` so that NTTs and other heavy operations automatically utilise all cores. For test builds (e.g., `allow_insecure`), parallelism can be disabled via feature flags if needed.

8. **Implement parameter-generation API**

9. Expose a function such as `FHEConfig::custom_depth(security: u32, depth: u32, t: u64) -> Result<Self>` that computes the polynomial dimension `n`, generates a modulus chain of appropriate primes, and sets the error distribution `eta` automatically. This could rely on heuristic formulas for noise growth and LWE security estimation.

10. **Add comprehensive tests and property-based testing**

11. Use the `proptest` crate (already in the dev-dependencies) to generate random plaintexts and parameter configurations and verify that encryption/decryption preserves the message and that homomorphic operations are correct modulo `t` for a variety of depths.

12. Add tests for `shadow-entropy` (checking that deterministic seeds yield the same ciphertexts) and for Galois rotations and neural evaluator functions.

13. **Improve documentation and comments**

14. Expand module-level comments in `ops/rns_fhe.rs` to include a high-level description of K-Elimination and show pseudocode for rescaling. Provide references to the underlying mathematical papers.

15. Document the expected runtime and noise overhead of each operation. This helps users design circuits that stay within the noise budget.

16. **Explore hardware acceleration**

17. Investigate using `std::simd` to implement vectorized modular arithmetic for NTTs and polynomial multiplications. This can provide significant speedups on modern CPUs without requiring unstable features.
18. Provide a CUDA or OpenCL back-end (behind a feature flag) for users who wish to run FHE computations on GPUs.

Conclusion

The NINE65-v5 library already delivers a powerful bootstrap-free FHE implementation with many innovative features. The existing tests in `lib.rs` and the `fhe_demo` program confirm the correctness of basic BFV operations, but some aspects—particularly dual-RNS multiplication and overflow handling—require further attention. Implementing the enhancements outlined above would improve reliability, performance and usability while maintaining the principles of **integer-only cryptography** and **post-quantum security** espoused in the project documentation [13]. Additional tests and documentation will also make the library more approachable to new users and researchers.

1 2 3 4 5 6 7 10 11 13 lib.rs

https://github.com/Skyelabz210/NINE65-v5/blob/main/crates/nine65/src/lib.rs

8 9 fhe_demo.rs

https://github.com/Skyelabz210/NINE65-v5/blob/main/crates/nine65/src/bin/fhe_demo.rs

12 Cargo.toml

https://github.com/Skyelabz210/NINE65-v5/blob/main/crates/nine65/Cargo.toml