Testability of Post-Quantum Cryptographic Algorithms

atsec Bootcamp
2/27/24
Chris Celi, CAVP Program Manager
christopher.celi@nist.gov

National Institute of Standards and Technology
U.S. Department of Commerce

Outline

- About the Cryptographic Algorithm Validation Program (CAVP)
- About the Automated Cryptographic Validation Test System (ACVTS)
- Post-Quantum Cryptography at NIST
- Validation testing on ‘new’ algorithms

Cryptographic Algorithm Validation Program

NIST

Automated Cryptographic Validation Testing System (ACVTS) provides automated validation testing of approved security functions and sensitive security parameter (SSP) generation and establishment methods.

Approved *(i.e, FIPS-approved and NIST Recommended) security functions and SSP generation and establishment methods for FIPS 140-3 are found in SP 800-140Cr1 and SP 800-140Dr1.*

- ACVTS Prod (2019) used by accredited labs to conduct validation testing.
- ACVTS Demo (2017) is a sandbox-style environment for anyone to request access and test.
- Over 2.3M vector sets served between Demo and Prod.
- 17ACVT scope open to first-party test labs, see NIST Handbook 150-17.
- Source code at https://github.com/usnistgov/ACVP-Server

💻 csrc.nist.gov/projects/cryptographic-algorithm-validation-program

Cryptographic Algorithm Validation Program NIST

- Goal: achieve two major assurances

Correctness

- Given a set of inputs, can the implementation generate the expected outputs
- Randomly generate inputs, compare against a reference implementation output

Security

- Does the implementation differ from the standard in any way that compromises the security assertions of the algorithm
- Target tests towards areas of weakness

ACVTS

NIST

- Open source Gen/Vals
  - C# code used to generate and validate test vectors
  - Continuously improved by the CAVP
  - https://github.com/usnistgov/ACVP-Server

- Offers a work bench to constantly improve the level of assurance

- CAVP goal is to introduce Demo testing for draft algorithm standards, to enable Prod testing once the standard is published

Post-Quantum Cryptography

- NIST started the Post-Quantum Cryptography Standardization effort in 2016 with a call for proposals

- Three draft standards have been published from these proposals with more to come soon

- Draft FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)
- Draft FIPS 204, Module-Lattice-Based Digital Signature Standard (ML-DSA)
- Draft FIPS 205, Stateless Hash-Based Digital Signature Standard (SLH-DSA)

- Full publications expected mid-2024

Post-Quantum Cryptography

- ML-KEM
  - Key Generation, Encapsulation, Decapsulation
- ML-DSA
  - Key Generation, Signature Generation, Signature Verification
- SLH-DSA
  - Key Generation, Signature Generation, Signature Verification

ML Key Generation

- Handled similarly for both ML-KEM and ML-DSA
- Get a random 256-bit seed*
- Expand it to the number of needed bits*
- Generate a number of vectors and matrices, the key pair*

ML Key Generation

- Get a random 256-bit seed*
  - *Generated from a deterministic random bit generator (DRBG)*
  - *Is the seed able to be provided as input to the function?*

- Expand it to the number of needed bits*
  - *How many bits are needed?*

- Generate a number of polynomial vectors and matrices, the key pair*
  - *Values are constrained by a modulo, how do we ensure uniformity?*

ML Key Generation

- Must require that the seed is able to be taken as input
- Random 256-bit seed, expanded using SHAKE

Correctness

- Generate random seeds, and expected keys
- Test implementation must generate the exact key

Security

- Impossible to determine the seed based on the generated key
- As long as every seed is allowed, there should not be an issue

ML Key Generation Rejection Sampling

- Uniform random values over an odd range
- Using bytes, we need a random [0, q] for some prime q
- Sample the bytes randomly, but reject the bytes if the value is out of the desired range
- Use SHAKE as a pseudorandom function, and continue requesting bytes until we have all the random values we need

ML Key Generation Rejection Sampling

- ML-DSA
  - Half byte to generate [-2, 2] or [-4, 4]

-2, -1, 0, 1, 2 = 5 total values	-4, -3, -2, -1, 0, 1, 2, 3, 4 = 9 total values
4 bits = 16 total values	4 bits = 16 total values
2 – (r mod 5), unless r = 15	4 – r, unless r >= 9
15/16 successes, 1/16 rejections	9/16 successes, 7/16 rejections

ML Key Generation

- Find a seed that leads to as many rejections as possible
- Sequence of half-bytes $(r_1, r_2, r_3\ldots)$ = SHAKE(seed)

Correctness

- Does the implementation handle the average number of rejections?
- Random seeds, over a number of test cases

Security

- Does the implementation handle the *worst case* number of rejections?
- Well, we can mine some Bitcoin…

ML Key Generation

- Need to find seed, where SHAKE(seed) = `0xFFFFFFFF`…
- Can only try every possible seed, and store useful results to be used on-demand in testing
- How many rejections is enough?
- Similar for ML-KEM, where the range is [0, 3329] sampled from 12 bits, 4096 possible values

ML-DSA Signatures

- Also uses rejection sampling on the signature generation
- Verification has several rejection criteria

Correctness

- Can an implementation generate <u>a</u> correct signature for given inputs?
- Can an implementation generate <u>the</u> correct signature for given inputs?

Security

- Can an implementation handle many rejections?
- Are all checks used when verifying a signature?

ML-DSA Signatures – “A” versus “The”

NIST

A signature

- Provide some inputs to the client
- Run Signature Verification to see if the signature is valid
- Allows greater flexibility for the randomized variant

The signature

- Provide all inputs to the client
- Compare the generated signature to the expected signature
- Allows testing of specific edge cases

- Testing will likely include both

ML-DSA Signature Verification

- Several potential reasons to reject a signature
- Keys are byte-strings, concatenations of several encoded values, each can be tested
- Relatively easy to modify specific bits in a signature or key

ML-KEM Encapsulation/Decapsulation

- Two important values, shared key K and ciphertext c
- Encapsulating party generates K and locks it in c
- Decapsulating party unlocks c to find K
- Loosely similar to Signature Generation and Verification
- Lots can go wrong while decapsulating a value but Decapsulation will always return something that looks like K
  - "Implicit rejection"

ML-KEM Encapsulation/Decapsulation

- Encapsulation – similar discussion to ML-DSA SigGen
  - <u>An</u> encapsulation versus <u>the</u> encapsulation

- Decapsulation – similar discussion to ML-DSA SigVer
  - Modifying parts of the key and ciphertext to trigger various failure conditions
  - Compare using the implicit rejection values rather than true/false

ML-KEM Decapsulation

- Decapsulation uses the same internal `K-PKE.Encrypt()` function that encapsulation uses
- This function is directly tested with encapsulation tests
- What if the implementation only uses decapsulation?
- Decapsulation would not directly check the results of `K-PKE.Encrypt()` are correct
- Potential component tests for the internal function necessary

Conclusion

NIST

Questions?

CAVP Program Manager

Chris Celi

christopher.celi@nist.gov

Tell us about the cool things you're testing with ACVTS!

Want to contribute? See our GitHub https://github.com/usnistgov/ACVP-Server