Systematic Testing of Lightweight Cryptographic Implementations [Extended Abstract]

Sydney Pugh, M S Raunak, D. Richard Kuhn, and Raghu Kacker

***Index Terms*—Cryptographic Algorithm, Lightweigh Cryptography, Metamorphic Testing.**

I. Introduction

With the ubiquity of the Internet and the proliferation of Internet of Things (IoT), the need for securing the communication amongst IoT and other similar devices has become extremely important. The National Institute of Standards and Technology (NIST) initiated the Lightweight Cryptography (LWC) project with the aim of standardizing cryptographic algorithms that are especially designed for such resource-constrained environments [1]. As candidate algorithms are submitted for consideration, it is important to test their implementations for potential bugs, which can be challenging.

Cryptographic algorithms are generally more complex, and their code is usually highly compact with intricate bit-manipulations. Moreover, there is often no easy-to-develop test oracle present to determine whether the outputs of the programs are correct or not. Together these attributes make it extremely challenging to run tests and to discover bugs in them. Structural coverage based testing approaches such as statement or branch coverage are typically not very effective in discovering bugs in these types of programs.

Building on our earlier work of systematically testing different cryptographic algorithms, we study the applicability of our test design approach over the LWC candidates' implementations.

II. Approach

Since cryptographic programs are sometimes labeled as "non-testable" due to the lack of easy-to-develop test oracles, designing effective test cases for them becomes a major source of difficulty. We have had previous success [2] in designing test cases based on the properties of the underlying algorithms that must be true under all circumstances. We apply similar strategies in designing the following tests for the LWC submissions.

1) Bit Contribution for Plaintext Test: Hash functions and Authenticated Encryption with Associated Data (AEAD) schemes are required to be second-preimage resistant. That is, given a message m and cryptographic function f, it should be extremely difficult to find a second preimage $m' \neq m$ such that $f(m') = f(m)$. Thus we can infer that every bit of the message must contribute towards the hash or encryption; otherwise, it would be easy to flip a bit and generate an encrypted message or message digest that produces the same hash or encryption. The Bit Contribution for Plaintext test is designed on this idea. For HASH algorithms, we generate a fixed message m. Then we compute the hash $h(m)$. Next we systematically flip a single bit at a time of m, producing m', then compute the hash $h(m')$. We store all the hashes in a table and search for collisions. If no collisions are found, then the test is passed. This test can also be applied to AEAD algorithms. For AEAD algorithms, we follow the same steps as for hash algorithms and just keep the associated data, nonce, and key constant for all encryptions.

A. Bit Contribution for Nonce Test

AEAD algorithms are expected to be secure when a nonce is not repeated under the same key. Hence the encryption of a message given two different nonces should be very different. The Bit Contribution for Nonce test observes what happens to the ciphertext as bits in the nonce change. For this test we aim to produce a matrix of dimensions nonce-bits by ciphertext-bits. Each value in the matrix tells you, out of x trials, how many times when you change the i^{th} bit of the nonce the j^{th} bit of the ciphertext changes. To fill the matrix we do the following. We generate a random plaintext pt, associated data ad, nonce n, and key k. Then we encrypt pt with ad, n, and k producing ciphertext ct. Then for each bit $i = 0...8 * |n|$, we flip bit i of n, re-encrypt pt, then increment all matrix values m_{ij} such that the j^{th} bit of ct and ct' are different, We repeat this 10,000 times (i.e., 10,000 trials).

For a random cipher, we expect matrix values to be close to $x/2$. Statistically, we expect to see a binomial distribution, $Binomial(x, 1/2)$. We can approximate this with a normal distribution as it is easier to calculate, $Normal(x/2, x/4)$. For $x = 10,000$ trials we expect matrix values to be in the interval $4750 < m_{ij} < 5250$ with high confidence. We expect a few matrix values to fall outside the interval with low probability, so matrix values *close* to the interval bounds are flagged but not considered as failures. However, if any $m_{i,j}$ falls significantly far outside of the interval, we call this a test failure.

B. Bit Contribution for Key Test

The Bit Contribution for Key test is designed the exact same as the Bit Contribution for Nonce test, except instead we look

S. Pugh and M. Raunak are with the Department of Computer Science, Loyola University Maryland, Baltimore, MD 21210. E-mail: sfpugh@loyola.edu, raunak@loyola.edu

R. Kuhn and R. Kacker are with the National Institute of Standards and Technology. E-mail: kuhn@nist.gov, raghu.kacker@nist.gov

at how the ciphertext changes as the key bits are changed. Hence our matrix is now of dimensions key-bits by ciphertext-bits. Each value in the matrix tells you out of x trials, how many times when you change the i^{th} bit of the key the j^{th} bit of the ciphertext changes. The procedure for filling the matrix is the same but we systematically flip each bit of the key $i = 0...8*|k|$ and keep the nonce constant. Similarly to Bit Contribution for Nonce, a matrix value that falls significantly far outside the interval $4750 < m_{ij} < 5250$ indicates a test failure.

C. Bit Exclusion Test

For HASH and AEAD algorithms, only the precise length of the message should be used by the cryptographic function. Thus, bits beyond the specified message length should be ignored. The Bit Exclusion test is designed to verify this property. For HASH algorithms, we generate a random message m of length n bytes. Then we compute the hash $h(m)$. Next, we systematically flip a single bit of m at a time in the four bytes after n (i.e., the $(n+1)^{th}$, $(n+2)^{th}$, $(n+3)^{th}$, and $(n+4)^{th}$ byte) and re-generate the hash, $h(m)'$. Since no bit hash been changed within the specified length of m, the hash should remain the same. If we find that $h(m) \neq h(m)'$, then the test is failed. This test can also be applied to AEAD algorithms, for which we follow the same procedure as for hash algorithms and just keep the associated data, nonce, and key constant for all encryptions.

D. Buffer Check Test

In the case that a ciphertext is invalid, the decryption-verification function of and AEAD algorithm should fail to return the plaintext. Therefore we can infer that every bit in the ciphertext should be considered by the decryption-verification function. Otherwise, it would be easy to flip a single bit in the ciphertext and deceitfully retrieve the plaintext. The Buffer Check test utilizes this idea. We generate a random plaintext pt, associated data ad, nonce n, and key k. Then we encrypt pt with ad, n, and k to produce a ciphertext ct. Now we systematically flip a single bit of ct at a time, producing ct', and then attempt to decrypt and verify ct' with ad, n, and k. If the decryption-verification function claims ct' is valid (i.e., function returns zero), then the test is failed. Also, if the plaintext buffer of the decryption-verification function contains a ten byte consecutive match to pt anywhere in the buffer, then the test is failed.

E. Ciphertext Length Check Test

The Ciphertext Length Check test verifies that the ciphertexts produced by an AEAD algorithm are of appropriate length. The API for the LWC competition specifies that algorithm implementations must contain a definition of variable $CRYPTO\_ABYTES$ which indicates that the ciphertext is at most $CRYPTO\_ABYTES$ bytes longer than the plaintext. Thus we derive a relation such that the length of any ciphertext must be longer than the plaintext length and shorter than the sum of the plaintext length and $CRYPTO\_ABYTES$. The test design is as follows. We generate a random plaintext pt, associated data ad, nonce n, and key k. Then we encrypt pt with ad, n, and k to produce a ciphertext ct. Then we check if $|ct| > |pt|$ and $|ct| < |pt| + CRYPTO\_ABYTES$. If true, then the test is passed. For this test, we vary the plaintext length from 0 to 256 bytes.

III. Results

A. AEAD Algorithms

For the AEAD algorithms we ran six tests: Bit Contribution for Plaintext, Bit Contribution for Nonce, Bit Contribution for Key, Bit Exclusion, Buffer Check, and Ciphertext Length Check. 57 algorithms were submitted to the LWC competition, and 56 of those submissions were considered complete enough to advance to round one. All 56 round one candidates provided an AEAD algorithm. Some candidates provided more than one implementation of their algorithm. 157 total reference implementations were considered in our testing experiment. Out of the 157 implementations, 0% failed the Bit Contribution for Plaintext test, 5.09% failed the Bit Contribution for Nonce test, 3.82% failed the Bit Contribution for Key test, 0% failed the Bit Exclusion test, 64.97% failed the Buffer Check test, and 1.91% failed the Ciphertext Length Check test.

B. HASH Algorithms

For the HASH algorithms we ran two tests: Bit Contribution for Plaintext and Bit Exclusion. Of the 56 round one candidates mentioned previously, 22 of those provided a hash algorithm. A total of 39 reference implementations were considered in our testing experiment. Out of the 39 implementations, none failed the Bit Contribution for Plaintext test or the Bit Exclusion test.

IV. Discussion

In this research, we have applied a systematic testing approach to lightweight cryptographic algorithm implementations. Since cryptographic code is often difficult to test due to code complexity and lack of a test oracle, we designed our test cases based on cryptographic properties that these implementations should satisfy. We have observed several test failures; and consequently, have identified bugs in some implementations. Our results suggest that this testing approach is effective at uncovering implementation failures in cryptographic code.

References

[1] NIST, "Submission requirements and evaluation criteria for the lightweight cryptography standardization process," August 2018. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf

[2] N. Mouha, M. Raunak, R. Kuhn, and R. Kacker, "Finding bugs in cryptographic hash function implementations," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 870–884, July 2018.