Testing Post Quantum Cryptographic Implementations for Constant-Time Execution

Joshua Koyeerath
Volgenau School of Engineering
George Mason University
Fairfax, Virginia 22032
Email: jkoyeera@gmu.edu

Omkar Kanore
Volgenau School of Engineering
George Mason University
Fairfax, Virginia 22032
Email: okanore@masonlive.gmu.edu

***Abstract*—Post Quantum Cryptography (PQC) is aimed to replace existing cryptosystems due to the threat of quantum computers. Researchers and organizations have come together to build libraries on open-source platforms which have implemented PQC algorithms. Our project aims to detect whether PQC implementations have timing leakages. In this paper we describe a method to test whether PQC implementations execute in constant time or not. We use a tool called as *dudect* to do the same. We run our test files written in C on Linux systems. Also, as part of our extended work, we have tested these implementations against open source static code analysis tools to find bugs and vulnerabilities.**

***Index Terms*—Post Quantum Cryptography, t-Testing, Timing attack, KEM, Signature, liboqs, libpqcrypto, dudect**

I. Introduction

With the development of quantum technology, a big change will inevitably occur in the field of encryption algorithms. Quantum computers take advantage of quantum physics and can perform calculations that ordinary computers are incapable of. This means that quantum computers can easily break through classic encryption algorithms such as RSA and ECC. Therefore, the emergence of post quantum cryptography is necessary.

The competition of NIST (National Institution of Standards and Technology) for PQC standards was officially launched in 2016. NIST focuses on the collection of the following three types of post-quantum cryptographic systems: encryption, key exchange, and digital signature [2]. Currently in Round 2 of the competition, these candidates can be divided into the following families:

- Multivariate-based
- Hash-based
- Isogeny-based
- Lattice-based
- Code-based

In order to test the implementations of PQC, we chose liboqs and libpqcrypto libraries. Figure 1 shows in detail which algorithms are implemented in the above mentioned libraries.

So, now we need to consider whether or not these PQC implementations are secure against timing attacks. Timing Attack is a type of Side Channel Attack about which we will discuss more in Section 2. Research has shown that timing attacks are effective in revealing secret/hidden data and they have broken several algorithms with variable-time implementations, such as SHA, TLS, RSA, and Diffie-Hellman [4].

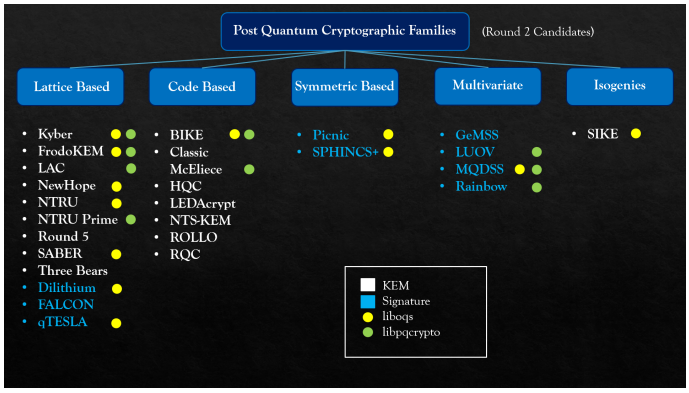

Fig. 1. Round 2 Candidates in liboqs and libpqcrypto

The following sections are organized as follows: Section II gives a short description about Side Channel Attacks and hence will define the scope of our project. In Section III we will describe the PQC libraries that were under consideration for testing. Then in section IV we will discuss about our testing methodology and give an overview of *dudect* with it's mathematical and software background. Section V will be devoted to the results that we obtained after testing and the inferences we can draw from these results. In section VI, we will discuss our findings after testing against open source static code analysis tools. Then finally, in Section VII, we will conclude with future work.

II. Side Channel Attacks

A side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself. Hence these attacks are independent of the theoretical/mathematical strengths that an algorithm may have but depends on the way in which the algorithm is implemented.

Below we list and describe some types of side-channel attacks [3]:

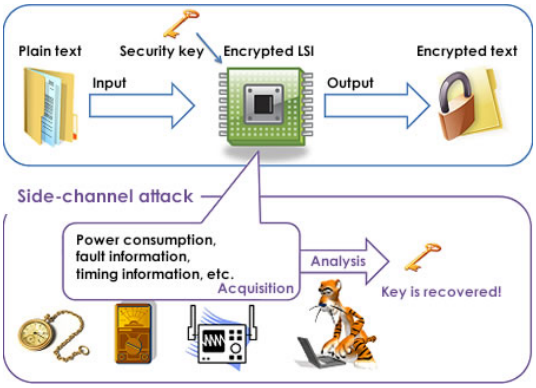

Fig. 2. Side Channel Attack

- Timing Attack
- Cache Attack
- Power-Monitoring Attack
- Electromagnetic Attack
- Acoustic Cryptanalysis
- Differential Fault Analysis
- Software-Initiated Fault Attacks

A. Cache Attacks

In this type of attack, the attacker monitors all cache accesses done by the victim program. This can lead to leakage of sensitive data that the program would load in the cache. There are many types of cache attacks like: Flush + Reload attack, Evict + Reload attack and Prime + Probe attack. In all the types, a fast-cache-access will tell the attacker that there was cache access in a particular location and slow-cache-access indicates otherwise.

B. Power and Electromagnetic Attacks

Power Attacks also called as Power-Monitoring Attacks happen when an attacker monitors the power consumption of a microprocessor while certain instructions are performed. It has two types: Simple Power Analysis (SPA) and Differential Power Analysis (DPA). In SPA, the attacker tries to find a simple correlation between power consumption and sections of code. In DPA, the attacker uses statistical and data analysis methods rather than sheer observational analysis. DPA is much harder to prevent than SPA. In case of Electromagnetic Attacks, the attacker captures the electromagnetic radiations emitting from a device and performs signal analysis. It also is divided into Simple and Differential Electromagnetic Attacks, SEMA and DEMA respectively. DEMA attacks are more complex and also but effective compared to SEMA.

C. Timing Attacks

In Timing Attacks, the attacker measures the time it takes to compute a piece of code. If a code is vulnerable to timing attacks, it may have variable execution time depending on the input that the code is dealing with. These measurements can be fed into a statistical model to infer the secret/hidden data that a program might use especially if it is a cryptographic algorithm. Article [3] provides an good example about how to exploit key based on the input of the microprocessor. As mentioned before, our project focuses only on testing resistance to timing attacks.

III. PQC Libraries

A. liboqs

liboqs is part of the Open Quantum Safe (OQS) project[6] led by Douglas Stebila and Michele Mosca. It aims to develop and integrate into applications quantum-safe cryptography to facilitate deployment and testing in real world contexts. In particular, OQS provides prototype integrations of liboqs into TLS and SSH, through OpenSSL and OpenSSH[7].

B. libpqcrypto

libpqcrypto[8] is a cryptographic software library produced by the PQCRYPTO project. libpqcrypto includes software for 77 cryptographic systems from 19 to 22 PQCRYPTO submissions. libpqcrypto collects this software into an integrated library, with a unified compilation framework, an automatic testing framework and command-line benchmarking tools.

IV. Testing PQC Libraries

A. Introduction to dudect

dudect (pronounced “dude”,“ct”) is a tool that is used test whether a program runs in constant-time or not[9]. Their approach is based on leakage detection tests. The tool relies on statistical analysis of execution time measurements. This allows the results to be somewhat independent of the underlying hardware infrastructure.

B. Math behind dudect

dudect uses Welch’s t-test to calculate timing variations. A t-test is a statistical hypothesis testing technique. For dudect, the hypothesis is that two populations/classes have equal means. If the hypothesis fails, then we say that there may be timing leakage. In order to test if this hypothesis holds true, we first calculate the value of t given by:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

If the value of t goes beyond a certain threshold (t_threshold), then the hypothesis fails. In the paper which describes *dudect*[1], t_threshold was set to 4.5 and in the code available on github, the value was set to 10. Here are two graphs mentioned in [1] showing two implementations of AES: one non-constant-time in figure 3 and one constant-time in figure 4.

There are many ways to select the two populations for testing, but *dudect* uses the fixed vs random class type. The random class by definition has random inputs and the fixed class has it’s inputs all set to zero. Once these two classes are prepared, the tests are conducted and appropriate t values are calculated to ensure if the hypothesis is true.

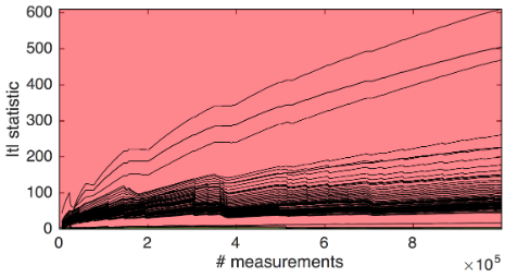

Fig. 3. Non Constant-time implementation of AES

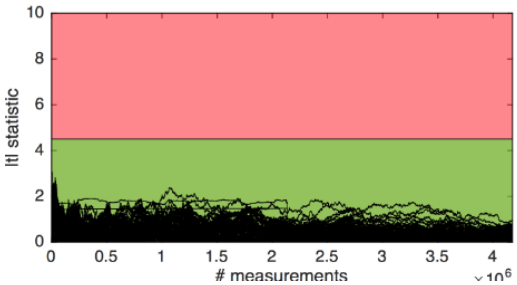

Fig. 4. Constant-time implementation of AES

C. Software behind dudect

The flowchart in Figure 5 describes the control flow of *dudect*.

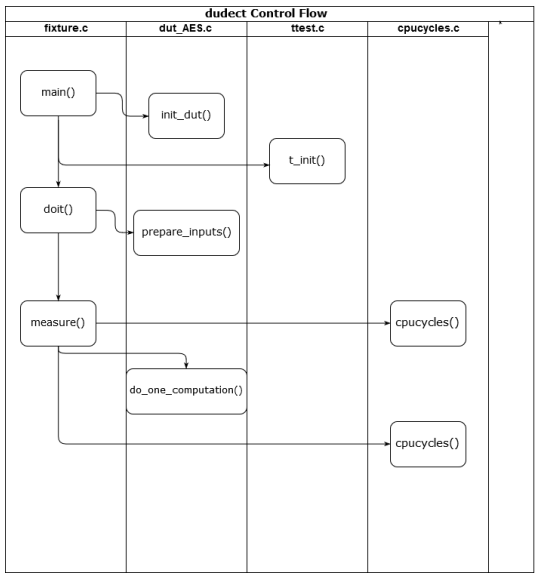

Fig. 5. dudect control flow

From the flowchart, `fixture.c` in the first column, is where all the main functions are called. The second column `dut_AES.c` is the test file that we have to create for each implementation (in this case, it is AES). The recording of measurements begin with calling of the `main()` in `fixture.c`. This function then calls `init_dut()` which is an initialization function written in the test file for any initialization needed, for example, creating public and private keypair. After that it calls the `t_init()` which initializes values for the t-test. Once all the values are set, `main.c` calls the `doit()` which is in an infinite loop. The `doit()` calls the `prepare_inputs()` in order to create and initialize the two populations as shown.

```
void prepare_inputs(uint8_t *input_data, uint8_t *classes) {
  randombytes(input_data, number_measurements * chunk_size);
  for (size_t i = 0; i < number_measurements; i++) {
    classes[i] = randombit();
    if (classes[i] == 0) {
      memset(input_data + (size_t)i * chunk_size, 0x00, chunk_size);
    } else {
      // leave random
    }
  }
```

Fig. 6. dudect: prepare_inputs function

`prepare_inputs()` creates a huge array called as `input_data[]` of the size of (*number_measurements* * *chunk_size*) taken from figure 7 and then fills it with random data. Then it selects certain chunks in the array and then fills it with zeros. Once the array is ready with both the populations, `doit()` calls `measure()`. `measure()` records the start and end time using `cpucycles()` and in between calls the `do_one_computation()`. `do_one_computation()` is the function in which we write a call to the encryption/decryption functions specified by an algorithm in the PQC library. The code snippet is shown in Figure 7

```
static uint32_t rk[44] = {0};
const size_t chunk_size = 16;
const size_t number_measurements = 1e6; // per test

uint8_t do_one_computation(uint8_t *data) {
  uint8_t in[16] = {0};
  uint8_t out[16] = {0};
  uint8_t ret = 0;

  memcpy(in, data, 16);

  // chain some encryptions
  for (int i = 0; i < 30; i++) {
    rijndaelEncrypt(rk, 10, in, out);
    memcpy(in, out, 16);
  }

  ret ^= out[0];
  return ret;
}
```

Fig. 7. dudect: do_one_computation function

Another important point to note is that *dudect* uses TSC(Time Stamp Counter) register to record time. Figure 8 is the screenshot of `cpucycles()` where the assembly - language instruction `rdtsc` is called. Figure **??** is the screenshot of `measure()` where `cpucycles()` is used.

```
int64_t cpucycles(void) {
  unsigned int hi, lo;

  __asm__ volatile("rdtsc\n\t" : "=a"(lo), "=d"(hi));
  return ((int64_t)lo) | (((int64_t)hi) << 32);
}
```

Fig. 8. dudect: cpucycles function

In order to test the behaviour of the algorithms in different scenarios and in-turn bring more authenticity to the data, each algorithm has to undergo two types of tests.

- One-Key-Different-Inputs (OKDI)
- One-Input-Different-Keys (OIDK)

For OIDK, we do not use the `input_data[]` array but we call the respective algorithm's `keypair()` function repeatedly to ensure valid keys. Also, for both types of tests, we only call the `decaps()` for KEMs and `sign()` for signatures since these functions use the secret key.

V. Results

Figure 9 and figure 10 display the results for some PQC implementations having undergone the OKDI and OIDK tests. The graphs are plotted as measurements (in millions) vs the maximum t value. For the understanding of the reader, to generate the figure 9, it took 3 laptops running simultaneously for 5 days. In Figure 9, we see no algorithm has their max t value greater than 10 and hence we can say that these implementations may be constant-time. Similar can be said of the OIDK test.

However, one algorithm implementation has stood out of the lot as shown in Fig 11.

It is interesting to note that Dilithium (a signature PQC algorithm) turned out to be constant-time for OIDK test but displayed non-constant-time behaviour by a huge margin ($t \approx 500$) in OKDI test. To further confirm, we reduced the $chunk\_size$ (which indicates the message size) from 2044 bytes (length of signature) to 16 bytes and the results still stay the same.

VI. Extended Work: Static Code Analysis

Since the aim of our project is to secure implementations, we tested the libraries against open source static code analysis tools. The C language is infamous for bringing in many bugs and vulnerabilites in the code. We used two tools for testing and following is their description.

A. Flawfinder

Flawfinder is a tool that searches through C/C++ source code looking for potential security flaws. It produces a list of hits (potential security flaws, also called findings), sorted by risk; the riskiest hits are shown first. The risk level is shown inside square brackets and varies from 0, very little risk, to 5, great risk. Flawfinder tool supports the CWE and is officially CWE-Compatible. Hit descriptions typically include a relevant Common Weakness Enumeration (CWE) identifier in parentheses where there is known to be a relevant CWE for each line of code where the vulnerability is found. Our primary aim was to bring the medium and high severity hits to a lesser number.

B. CCPcheck

CCPcheck is a tool for static analysis of C/C++ code. It checks for memory leaks, mismatching allocation-deallocation, buffer overrun, and many more. It provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs.

C. Testing and Results

When flawfinder was used on liboqs library, we found a total of 6137 hits, where none of them were severe alerts. 5 lines of code had a high security risk of level 4 whereas 6128 lines of code had a medium security risk of level 2. When ran on libpqcrypto library, we found 4 lines of code had a high security risk of level 4 whereas 1317 lines of code has medium security risk of level 2. We analyzed these errors, visited the CWE website and conducted some research on these bugs and added a fix for it. This activity led to the reduction in the number of hits for High severity bug to almost half and medium severity bug by 30%. The result for this activity is depicted on the pie chart diagram for liboqs in Figure 12 and libpqcrypto library in Figure 13.

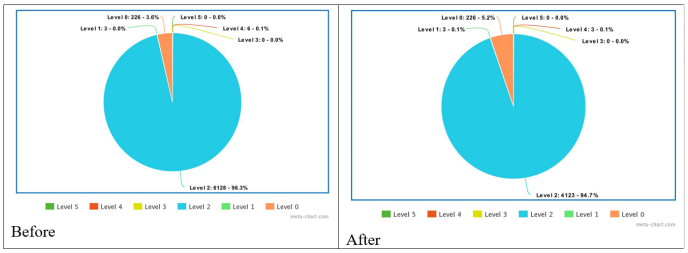

Fig. 12. Flawfinder: liboqs

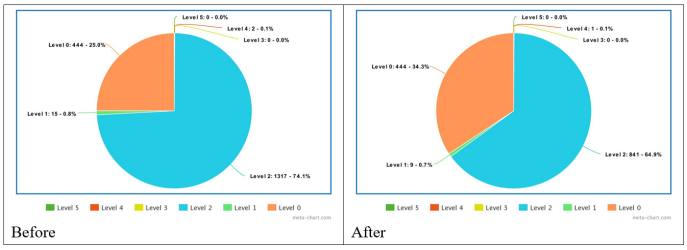

Fig. 13. Flawfinder: libpqcrypto

When ccpcheck tool was used on liboqs library, we found a total of 50 hits for vulnerabilities, where only 1 of them was of high-level risk. The pie chart in Figure 14 diagram provides a list of vulnerabilities, the number of instances in the project along with the percentage of each.

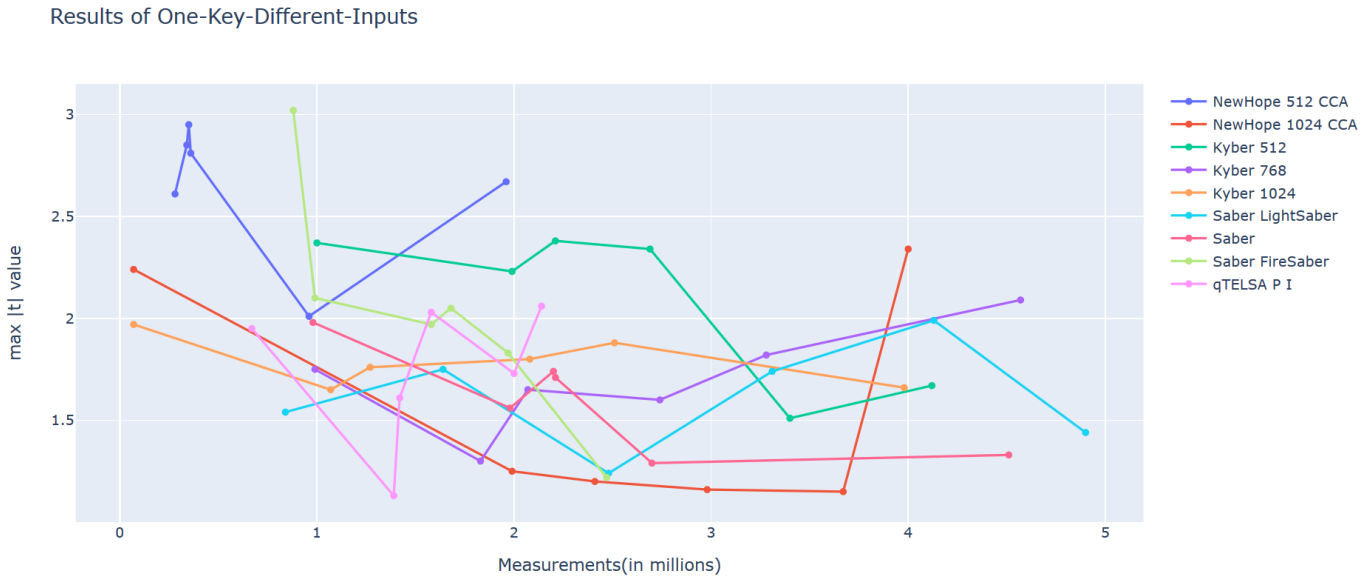

Fig. 9. Results for OKDI Test

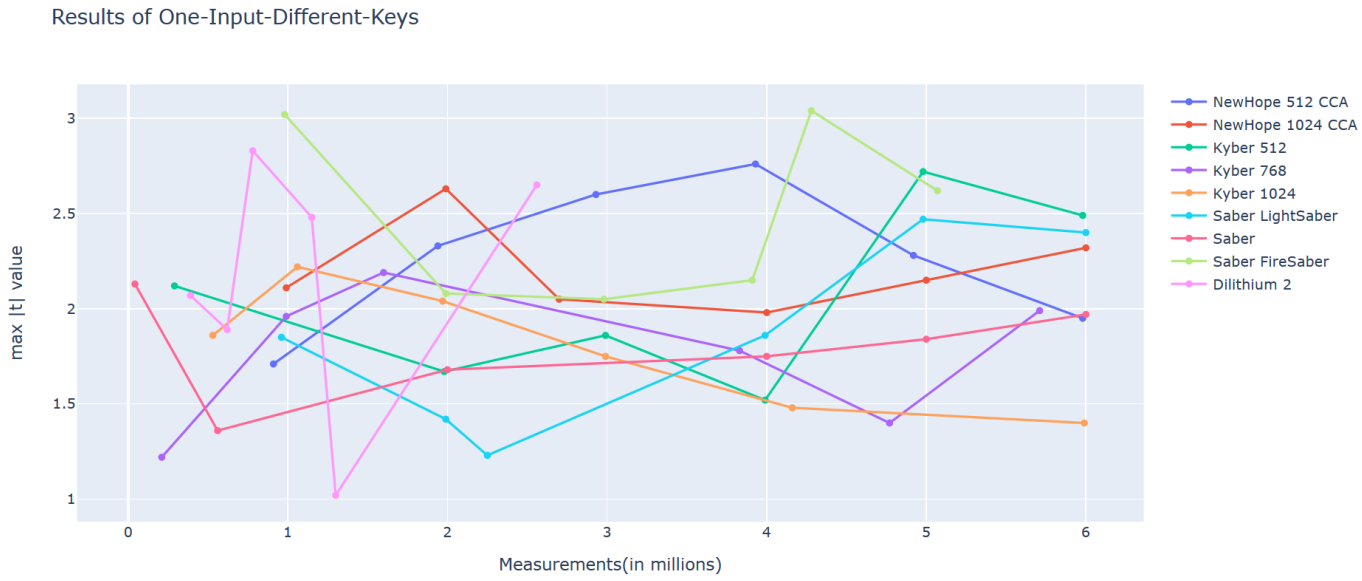

Fig. 10. Results for OIDK Test

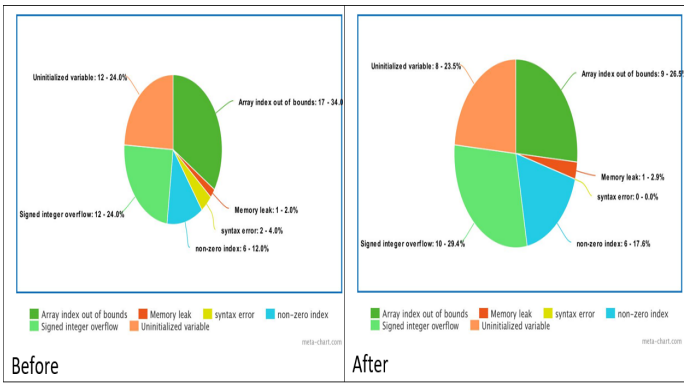

Fig. 14. CCPCheck: liboqs

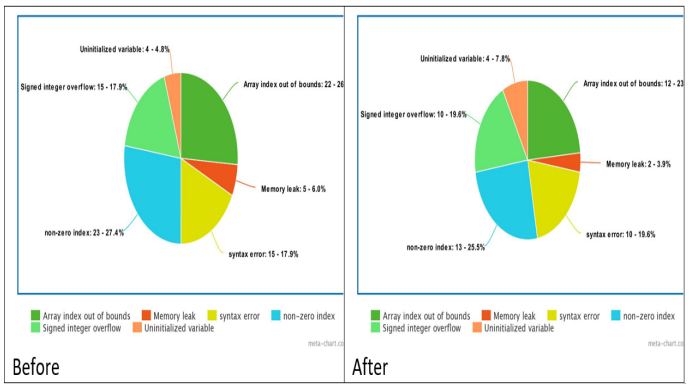

Fig. 15. CCPCheck: libpqcrypto

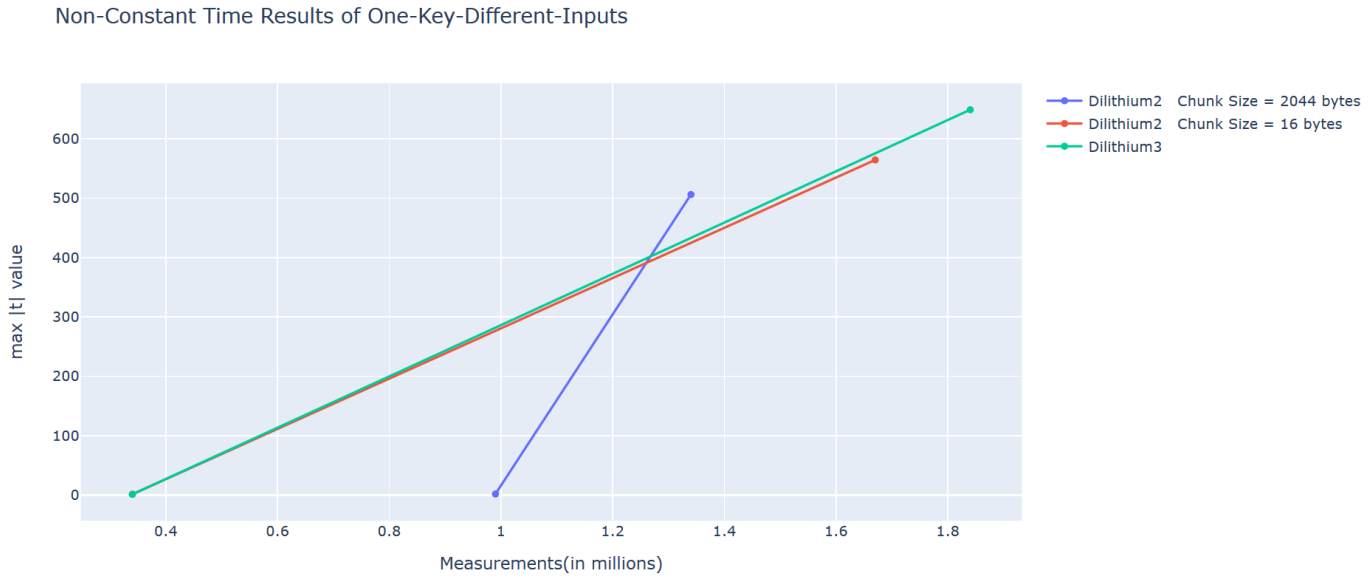

Fig. 11. Non constant results for OIDK Test

When ccpcheck tool was used on libpqcrypto library, we found a total of 84 hits for vulnerabilities, where only 5 of them was of high-level risk. The pie chart diagram in Figure 15 provides a list of vulnerabilities, the number of instances in the project along with the percentage of each. CCPcheck provides a report with line number, the path of the file and a note of potential vulnerability. This helped us identify the exact issue and find a fix for it. Iterating on various fixes we were able to bring down the syntax error issue for liboqs library to 0% and signed integer overflow error to 60%. For libpqcrypto library, we were able to bring the Memory leakage, Signed integer overflow errors down to 60%.

VII. Conclusions and Future Work

Since we have observed non-constant-time behaviour from Dilithium, work needs to be undertaken so that Dilithium becomes resistant to timing attacks and there is reduction in timing leakage. However, since only one tool was used to conduct leakage detection, we cannot say with certainty that other algorithms have constant-time implementations. Hence more work needs to be done using testing tools having different approaches in order to confirm our findings. The results that we see are tests done for the liboqs library. However test files have been created for libpqcrypto which just need to be executed to get results. In our extended work, we were able to find an fix most of the high-risk and medium-risk vulnerabilities. There are many more tools that can be used to perform an indepth analysis of code either statically or dynamically to lower the number of known vulnerabilites even more.

`liboqs` and `libpqcrypto` were created so that researchers/enthusiasts and even people in the industry who would like to experiment with the upcoming PQC algorithms could do so freely and incorporate these functions in their projects. Maintaining security in implementation hence also is of utmost importance. We believe that our work will help identifying any areas related to constant-time execution that the developers might have missed, hence securing the libraries from future potential attacks.

References

[1] O. Reparaz, J. Balasch and I. Verbauwhede, "Dude, is my code constant time?," Design, Automation& Test in Europe Conference Exhibition (DATE), 2017, Lausanne, 2017, pp. 1697-1702. doi: 10.23919/DATE.2017.7927267

[2] PQC Archive - Post-Quantum Cryptography, CSRC. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/PQC-Archive. [Accessed: 10-Dec-2019].

[3] J. Iriarte, Side Channel Attacks: What They Are and How to Prevent Them, Jungle Disk Blog, 11-Jul-2019. [Online]. Available: https://www.jungledisk.com/blog/2017/12/28/be-aware-of-side-channel-attacks/. [Accessed: 10-Dec-2019].

[4] P. C. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, Advances in Cryptology CRYPTO 96 Lecture Notes in Computer Science, pp. 104113, 1996.

[5] k0resh, Timing attacks part 1, The Latest News from Research at Kudelski Security, 13-Dec-2013. [Online]. Available: https://research.kudelskisecurity.com/2013/12/13/timing-attacks-part-1/. [Accessed: 10-Dec-2019]

[6] D. Stebila, M. Mosca, "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project," Selected Areas in Cryptography, 2017, Springer International Publishing. [Online]. Available: http://link.springer.com/10.1007/978-3-319-69453-5_2

[7] 'Open Quantum Safe', 2019. [Online]. Available: https://openquantumsafe.org/. [Accessed: 10- Dec- 2019].

[8] 'libpqcrypto', 2019. [Online]. Available: https://libpqcrypto.org/. [Accessed: 10- Dec- 2019].

[9] 'GitHub Repository: dudect', 2019. [Online]. Available: https://github.com/oreparaz/dudect. [Accessed: 9- Dec- 2019].