qTESLA: Practical Implementations of a Quantum Attack Resistant Signature Scheme

Michael Burger
Scientific Computing
TU Darmstadt
Darmstadt, Germany
michael.burger@tu-darmstadt.de

Juliane Krämer
QPC
TU Darmstadt
Darmstadt, Germany
juliane@qpc.tu-darmstadt.de

Christian Bischof
Scientific Computing
TU Darmstadt
Darmstadt, Germany
christian.bischof@tu-darmstadt.de

***Abstract*—Due to the advent of quantum computers, the security of existing public-key cryptography is threatened since quantum computers are expected to be able to solve the underlying mathematical problems efficiently. Hence, quantum resistant alternatives are required. Consequently, about 70 post-quantum scheme candidates were submitted to the National Institute of Standards and Technology (NIST) standardization effort. One candidate is the qTESLA signature scheme. We present an efficient shared-memory parallelization of qTESLA's core routines, analyze the speedup in-depth and show that it can compete with the two most commonly used signature schemes RSA and ECDSA which are quantum-vulnerable. The speed is further increased by semi-automatic tuning of qTESLA's configuration parameters based on results of multi-parameter performance models. We show how to considerably increase qTESLA's usability through the Java Native Interface (JNI) without performance penalty. The analysis on x86 and ARM architecture employing three operating systems demonstrates the achieved portability. The enhanced performance, its straight forward usability and the high portability of our implementation make it a quantum-safe replacement for the state-of-the-art schemes.**

***Index Terms*—SMP parallelization, performance modeling, software engineering, security, quantum resistance**

I. Introduction

Cryptography and, in particular, public-key cryptography (PKC) is indispensable for our everyday life. The security of currently used public-key schemes is based on the hardness of two mathematical problems: the factorization problem and the discrete logarithm problem. While these schemes withstand classical cryptanalysis, it is known since the mid-1990s that with Shor's algorithm running on a quantum computer, both the factorization problem and the discrete logarithm problem can be solved efficiently [1]. Hence, we need replacements for currently used PKC as soon as large quantum computers exist. These replacements are intended to be used on classical computers and hence have to provide the same functionality as classical PKC, but rely on mathematical problems which resist attacks with quantum computers. Cryptography that is based on mathematical problems which are assumed to resist those attacks is called post-quantum cryptography (PQC). Several types of PQC exist. The five most important types are lattice-based (e.g. Dilithium, Falcon), hash-based (e.g. SPHINCS$^+$), isogeny-based (e.g. Sike), code-based (e.g. FrodoKEM, ROLLO), and multivariate cryptography (e.g. GeMSS, LUOV). For further information about the listed examples and more schemes are available online [2].

Although no one knows when powerful enough quantum computers will be available, many estimates range between 10 and 20 years from now, e.g., [3]. Hence, the era in which we have to use PQC is approaching, and we have to start the transition from classical cryptography to PQC immediately. One avenue to promote the use of PQC is the standardization of post-quantum algorithms, since the standardization of cryptography makes the adoption of new schemes easier for practitioners. The best known PQC standardization effort is taking place at the US-American National Institute of Standards and Technology (NIST) [4].

The two most fundamental cryptographic primitives are encryption and digital signature schemes. While encryption is used to ensure confidentiality, digital signatures are used to provide authenticity and data integrity. As such, digital signatures protect, e.g., secure software updates, also in the context of high High Performance Computing (HPC). Hence, the research on post-quantum signatures is very active and many post-quantum signatures have been submitted to the NIST PQC standardization process. Most of these signature schemes belong to the family of lattice-based cryptography. One of these schemes is qTESLA [5], which made it to the second round of the standardization process. Building upon several predecessors, the scheme qTESLA combines the most relevant features of these and, hence, has several advantages, as compared to other schemes: the instantiations of qTESLA are provably secure. Despite this strong security argument, qTESLA is comparably efficient.

The main contributions of this work are:

- We present a parallelization of the core routines of qTESLA to increase its efficiency.
- We considerably increase qTESLA's usability and portability by employing Java JNI which are demonstrated with tests on different systems.
- We further reduce qTESLA's runtimes by semi-automatic parameter tuning with Extra-P.

- Altogether, our implementation of qTESLA can be a quantum-safe replacement for state-of-the-art schemes.

II. qTESLA Signature Scheme

Digital signature schemes consist of three algorithms: key generation, signing, and verification.

The lattice-based digital signature scheme qTESLA is designed to be easy to implement. A reference implementation is available [6]. The fact that Gaussian sampling - a subroutine commonly necessary for lattice-based cryptographic algorithms - is only used for key generation, simplifies, in particular, the signing and verification procedures. It also facilitates implementations that are protected against physical attacks, i.e., side channel and fault attacks. Moreover, the scheme supports constant-time implementations, which defeat, for instance, certain power analysis and cache attacks. The security of qTESLA is proven in the quantum random oracle model (QROM). This model explicitly considers quantum adversaries. The authors of qTESLA propose two parameter sets, qTESLA-p-I and qTESLA-p-III, cf. [5, Table 2], which are chosen according to the requirements of the NIST PQC standardization process [4], and which achieve around 139 and 279 quantum bit hardness, respectively. As their names suggest, these parameter sets correspond to NIST's security categories 1 and 3. An important characteristic of these parameters is that they are provably secure, i.e., as long as the underlying mathematical problem provides a certain bit hardness, the instantiation of the scheme achieves the declared bit security. While this is a strong security argument, usually provably secure instantiations of cryptographic schemes are less efficient than instantiations that are not provably secure. In the rest of this paper, we always refer to the strongest parameter set, i.e., qTESLA-p-III.

The security of qTESLA is based on the hardness of a variant of the learning with errors (LWE) problem. The LWE problem is a computational problem which finds a linear function given so-called samples which consist of slightly perturbed outputs of the function, i.e., some noise has been added to the correct outputs. It can be shown that the LWE problem is as hard as certain worst-case lattice problems, which makes it an attractive computational problem for post-quantum public-key cryptography. While the LWE problem originally is defined over the ring of integers modulo some q, for efficiency reasons nowadays usually the ring-LWE variant is used, which is the LWE problem working in polynomial rings over finite fields [7]. qTESLA is also based on ring-LWE. More concretely, the security of qTESLA is based on the so-called decisional R-LWE problem, which consists in distinguishing whether a given sample is an R-LWE sample or a sample from a uniform distribution. Thus, an adversary who can forge qTESLA signatures can also solve the decisional R-LWE problem.

Since qTESLA works in polynomial rings over finite fields and polynomial multiplication over a finite field is one of its fundamental operations, in both the key generation algorithm (cf. Algorithm 1) and in the signing procedure (cf. Algorithm 2) univariate polynomials of a predefined degree have to be sampled. Two kinds of polynomials are used. The polynomials sampled by GenA have uniform random integer coefficients (cf. Algorithm 1 line 4) and Algorithm 2 line 5). The polynomials sampled by using GaussSampler have discrete Gaussian distributed integer coefficients (cf. Algorithm 1 lines 6 -11). They are part of the secret key.

Algorithm 1 Key Generation. Taken from [5].

Require: -

Ensure: key pair (sk, pk) with secret key $sk = (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y, g)$ and public key $pk = (t_1, \ldots, t_k, \mathsf{seed}_a)$

1: $\mathsf{counter} \leftarrow 1$
2: $\mathsf{pre\text{-}seed} \leftarrow_{\$} \{0,1\}^{\kappa}$
3: $\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}, \mathsf{seed}_a, \mathsf{seed}_y \leftarrow \mathsf{PRF}_1(\mathsf{pre\text{-}seed})$
4: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$
5: **do**
6: $\quad s \leftarrow \mathsf{GaussSampler}(\mathsf{seed}_s, \mathsf{counter})$
7: $\quad \mathsf{counter} \leftarrow \mathsf{counter} + 1$
8: **while** $\mathsf{checkS}(s) \neq 0$
9: **for** $i = 1, \ldots, k$ **do**
10: $\quad$ **do**
11: $\quad\quad e_i \leftarrow \mathsf{GaussSampler}(\mathsf{seed}_{e_i}, \mathsf{counter})$
12: $\quad\quad \mathsf{counter} \leftarrow \mathsf{counter} + 1$
13: $\quad$ **while** $\mathsf{checkE}(e_i) \neq 0$
14: $\quad t_i \leftarrow a_i s + e_i \mod q$
15: **end for**
16: $g \leftarrow \mathsf{G}(t_1, \ldots, t_k)$
17: $sk \leftarrow (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y, g)$
18: $pk \leftarrow (t_1, \ldots, t_k, \mathsf{seed}_a)$
19: **return** $sk,\ pk$

In this work, we are concerned with optimizing the efficiency of qTESLA. We target, in particular, program loops during key generation (cf. Algorithm 1 lines 5 -8 / 10 - 13) and signing (cf. Algorithm 2 lines 4-20).

The loop in lines 5 - 8 in Algorithm 1 aims at generating a secret polynomial s with Gaussian distributed coefficients that has an additional property that is needed for the security of the scheme. This requirement is checked with the checkS function (cf. [5, Algorithm 2]). The check bounds the coefficients of the polynomial s and thereby the key space. It ensures $\sum_{i=1}^{h} \max_i(s) < L_S$ for a parameter L_S. Also in the second loop during key generation, polynomials with Gaussian distributed coefficients are sampled as well as an additional property which is related to the correctness of the signature scheme. It is checked with the checkE function (cf. [5, Algorithm 1]). The check bounds the coefficients of the error polynomials e_i which are part of the R-LWE samples. The check ensures $\sum_{i=1}^{h} \max_i(e) < L_E$ for a parameter L_E. Again, this bounds the key space of qTESLA.

Algorithm 2 qTESLA's Signature Generation. Taken from [5].

Require: message m, and secret key $sk = (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y, g)$
Ensure: signature (z, c')

1: $\mathsf{counter} \leftarrow 1$
2: $r \leftarrow_{\$} \{0,1\}^{\kappa}$
3: $\mathsf{rand} \leftarrow \mathsf{PRF}_2(\mathsf{seed}_y, r, \mathsf{G}(m))$
4: $y \leftarrow \mathsf{ySampler}(\mathsf{rand}, \mathsf{counter})$
5: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$
6: **for** $i = 1, \ldots, k$ **do**
7: $\quad v_i = a_i y \bmod^{\pm} q$
8: **end for**
9: $c' \leftarrow \mathsf{H}(v_1, \ldots, v_k, \mathsf{G}(m), g)$
10: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$
11: $z \leftarrow y + sc$
12: **if** $z \notin \mathcal{R}_{[B-S]}$ **then**
13: $\quad \mathsf{counter} \leftarrow \mathsf{counter} + 1$
14: $\quad$ Restart at step 4
15: **end if**
16: **for** $i = 1, \ldots, k$ **do**
17: $\quad w_i \leftarrow v_i - e_i c \bmod^{\pm} q$
18: $\quad$ **if** $\|[w_i]_L\|_\infty \geq 2^{d-1} - E \vee \|w_i\|_\infty \geq \lfloor q/2 \rfloor - E$ **then**
19: $\quad\quad \mathsf{counter} \leftarrow \mathsf{counter} + 1$
20: $\quad\quad$ Restart at step 4
21: $\quad$ **end if**
22: **end for**
23: **return** (z, c')

In the signing procedure, the signature is computed within the loop. Again, two requirements regarding security and correctness have to be fulfilled, and if one of them is not fulfilled, the signature generation starts again in line 4. The security check is computed in line 12 of Algorithm 2. It ensures that the coefficients of a part of the signature come from a restricted set. The correctness check is computed in line 18 of Algorithm 2.

III. Related Work

In the realm of post-quantum cryptography, many encryption and signature schemes have been proposed over the last few decades. Lattice-based cryptographic schemes make up the largest part of these proposals. Two lattice-based digital signature schemes - Dilithium [8] and FALCON [9] - are, together with qTESLA, especially relevant. As required by NIST, reference implementations for both schemes are available online [8], [9].

The advantage of Dilithium, as compared to qTESLA, is good, balanced efficiency with respect to space and time. However, the parameters of Dilithium are not provably secure and qTESLA promises to offer better protection against physical attacks than Dilithium. Dilithium proposes parameters according to NIST's security categories 1, 2, and 3. FALCON proposes three parameter sets which cover all five security categories from NIST.

Several basic building blocks of qTESLA were further investigated in related work.

For example, Discrete Gaussian Sampling as employed in qTESLA's key generation routine and as required in nearly all lattice-based security solutions [10]. Micciancio and Walter [10] presented algorithmic variants which are on the one hand generic, i.e., their parameter sets are configurable. On the other hand, they are more resilient to several kinds of side-channel attacks since a constant time implementation is possible. In recent work [11], Zhao et al. propose another scheme for Gaussian Sampling which also allows constant-time implementation while being compact. It is based on Rényi divergence and the polynomial approximation of transcendental functions. A branch of qTESLA was already employed as a use-case to demonstrate the applicability of this sampler.

A hotspot in qTESLA is the sparse polynomial multiplication which is of complexity $O(n \log n)$ with n being a bound on the largest degree. Cole and Hariharan presented a nearly optimal multiplication algorithm in the case that all occurring coefficients are positive in 2002 [12]. Recently, Nakos proposed a multiplication algorithm [13] that removes this limitation and includes non-positive coefficients. Its complexity is near-linear in the size of the sum of the input and the output.

Enhancements in those building blocks are complementary to the optimizations presented in this paper and, hence, can be integrated supplementary.

IV. Implementation and Optimization

All different versions of our qTESLA implementations explained in this section are available at https://github.com/tudasc/qTESLA.

A. Parallelization approach in C

Here, we describe the modifications applied to the original qTESLA code [14] and justify our decisions. The major challenge is that all routines already run very fast so that parallelization overhead may not be neglected.

a) Key generation: As explained in Section II, key generation is composed of two successive polynomial sampling routines where the first one samples five error polynomials and the second one the secret polynomial (cf. Algorithm 1).

The abortion of the first while-loop checkE is controlled by parameter L_E and checkS by L_S respectively. Code profiling reveals that the second while-loop always succeeds in very few attempts and only consumes a very small portion of the overall key generation time. Hence, the overhead of spawning the parallelization is higher than the resulting speedup. Consequently, we decided to execute the for-loop in line 9 in parallel and to distribute the random seeds across the different threads.

b) Signature generation: An abstract C-like sign algorithm is given in Listing 1.

```
generateRandmonessOfSecretKey(char* skey, [...]);
polynomial a = generatePolynomalFromKey(char* skey);
randomness r = randomnessByHashingMessage(char* me);
int seed = 0;
while (true) {
```

```
  polynomial y = sampleY(r, ++seed);
  polynomial ytt = numberTheoreticTransformNTT(y);
  polynomial v = multiplciation(a, ytt);

  ... hashing, encoding ...
  if (!testRejectionCriteria())    continue;
  ... polynimial arithmetic ...

  if (!testCorrectness()) continue;
  break;
}
return encodeSignature();
```

Listing 1. Abstracted signature generation algorithm

The while-loop randomly samples the polynomial besides deterministic operations that check if the polynomial can be employed for signature generation. We parallelize it by first executing all deterministic processes at function entry. Afterward, a parallel region is spawned which includes the while-loop and shares information about the random seed seed and the randomness. Each thread executes iterations of the while-loop and an asynchronous shared flag is employed to notify the other threads about termination in the case of success by another thread. If a thread succeeds, all others cancel their work after the current iteration or directly before the testRejectionCriteria()-call and perform an early exit. Profiling revealed that this point splits the workload of the loop in about equal parts. After the implicit barrier of the parallel region, the master thread takes the polynomials found by the last thread that sets the shared flag and encodes the signature.

c) Signature verification: In contrast to the two generation algorithms, the verification does not include random elements. An abstracted summary is given by Listing 2.

```
decodeSignature(char* sign);
if(!testCondition1)
  return false;

decodePublicKey(char* pkey);
... polynomial arithmetic ...

for (int i=0; i < 5; i++) {
  ... polynomial arithmetic with
  the error polynomials ...
}
// ... do shaking and hashing
if (! checkIfValuesMatchSignature())
  return false;
```

Listing 2. Abstracted verification algorithm of qTESLA

Our parallelization strategy is to distribute the work of the for-loop between the threads employed.

B. Java version

For comparison reasons, we consider a pre-existing qTESLA implementation from https://github.com/ue17apyt/qTESLA which is also part of the Bouncy Castle library [15]. The code tries to translate the C reference implementation directly. However, the available version works with a deprecated parameter set which was updated in [5, Table 1]. This also influences the efficiently implemented encode and decode routines for the signatures and the public and private keys. These are based on bit operations which depend on the parameter set. Hence, we updated all parameters within the code, re-implemented the encode and decode with the appropriate bit arithmetic, and unified the mapping of C to Java data types. We validated the correctness by exchanging and signing/verifying messages between the C reference version and our new Java implementation. We also tried to apply similar parallelization strategies as employed for C, but the overhead of Java Threads is too high and employing more than one thread always reduces the overall speed. Fully implementing qTESLA in a programming language focused on parallelization like Go is out of the scope of this paper.

C. JNI and JCA

Since the focus of this work is not only to improve the performance of qTESLA but also to enhance its usability, we also employ the Java Native Interface (JNI) [16], allowing to access the C-code as a shared library by Java code. Our interface provides all required methods to practically employ qTESLA as a stand-alone application or as a library. The keys, the signatures and the messages are passed as Java byte[] and automatically converted to the required format. This avoids the former usage of pointers and their explicit allocation with the appropriate size and their manual deletion. Hence, the usage is now less error prone and memory leaks are prevented. Sophisticated error-handling is also integrated through the novel interface. Additionally, the parameters of the underlying qTESLA implementation can be queried which is important if they change in the future. Furthermore, the degree of parallelism to employ can be set easily without employing environment variables.

To make the usage of qTESLA more straightforward, we employ the Java Cryptography Architecture (JCA) and implement qTESLA as a Cryptographic Service Provider (CSP). In that way, using qTESLA is realizable with only a few lines of Java code which is demonstrated in Listing 3.

```
byte[] message = createRandomMessage(length);
// Generate public and private key
KeyPairGenerator keyGenerator =
  KeyPairGenerator.getInstance("qTESLA-P3");
KeyPair kp = keyGenerator.generateKeyPair();

// Sign the message
Signature sig = Signature.getInstance("qTESLA-P3");
sig.initSign(kp.getPrivate());
sig.update(message);
byte[] signature = sig.sign();

// Verify the signature
sig.initVerify(kp.getPublic());
sig.update(message);
if(!sig.verify(signature))
  System.err.printf ( "Signature not verified!");
```

Listing 3. Using qTESLA as CSP

The JCA and the CSPs are in widespread use and their interface is well documented. Consequently, users already

familiar with the concepts can directly employ PQC or adapt existing JCA-code to its usage by just changing a few lines of code. Beginners is provided an easy entry point by the existing documentation.

D. Security Implications

We are not aware of any security vulnerabilities introduced by the parallelization or the JNI/CSP usage.

JNI is the de facto standard for addressing C code with Java and is used for decades for performant integration of native code into Java applications [17]. Additionally, it avoids the usage of Java third party tools and only employs the core functionalities of Java which have been proven to be very safe [18]. Potential security vulnerabilities, hence, would not only affect qTESLA, but any code that is implemented in C and addressed with Java JNI. To the best of our knowledge, such vulnerabilities are not known. The same applies to the CSP interface and all signature schemes implemented with it, respectively.

Regarding physical attacks, the best case from an attacker's point of view is to isolate a single thread and get side channel information for it, e.g., power or timing information. Then, however, the attacker has only as much information as he could gain from a serial implementation. Therefore, we are confident that our parallelization does not enable additional attack vectors. On the contrary, we believe that the parallelization improves the physical security, since significantly more effort is required from an attacker to get side channel information of a single thread.

V. Parameter Optimization

We also investigated the parameter set of our parallelized qTESLA C version. To that end, we employed a special version of qTESLA which allows to modify the parameters L_E and L_S. As explained in Section II, they influence the success rate for the sampling procedures in the key and the signature generation process. The dependent parameters of L_E and L_S are calculated by a SageMath script before compiling qTESLA.

To optimize the parameter configuration, we performed automated runs profiled with Score-P [19] which vary the parameters in a step size of 4 in the range $880 <= L_S = L_E < 940$ and create five measurements for each configuration. The whole key and signature generation and the verification function are instrumented. The measurements are stored as *cubex* files. A tool written in C++ extracts the runtime of the slowest thread for each instrumented function as indicator for its performance. In that way, we directly take our parallelization for modeling into account. The results are exported as *json* files in a special format for the performance modeling tool Extra-P [20].

Those *json* files can be used as input for Extra-P which is able to generate models that relate the runtime (i.e. also the runtime of our slowest thread) to both input parameters L_E and L_S. Following the model and visual inspection of the measurement points, increasing L_E and L_S decreases the runtime of key generation in the investigated range while decreasing them considerably increases the runtime.

For the signature generation, the model and the points show that increasing the parameter L_S will increase the sum of the runtime of all threads in the parallel region but the runtime of the slowest thread slightly decreases in the range $920 \leq L_S = L_E \leq 930$.

Since both parameters do not directly influence the verification, Extra-P can not create a consistent model and the points do not have a recognizable pattern.

From the above we deliver the prediction that setting $L_E = L_S = 929$ results in a faster signature scheme.

VI. Evaluation

Here, we perform an in-depth analysis of our qTESLA implementations with different compilers and operating systems. We compare its performance to RSA and ECDSA as the state-of-the-art schemes. Since qTESLA-p-III has a bit security of 160 bit, we employ RSA-6144 and ECDSA with the *SECP384R1*-curve which provide a comparable security level on classical hardware.

A. Methodology

a) Test systems: The x64 computer employed has an AMD Ryzen 9 3900, with 12 cores (24 threads) running at a base clock of 3.1 GHz and a turbo clock of up to 4.3 GHz. We intentionally keep turbo clock and simultaneous multi-threading (SMT) active to simulate real world situations although this may cause variations in the measurements. The system has 32 GB of main memory. It runs Windows 10 (Build 1909) and CentOS 7.2. For Windows, we employ the Microsoft Visual Studio 2019 C compiler and for Linux GCC 10.1 and Clang 10.0. Oracle javac is used in Windows (1.8.0_221-b11) and Linux (1.8.0_262). The OpenSSL library in version 1.1.1 is used for both systems compiled with its standard CMAKE Release x64 build.

The ARMv8-A system is a Xiaomi Redmi 8 Pro with a Mediatek Helio G90T SoC (2x 2.05GHz Cortex-A76 and 6x 2.00GHz Cortex-A55) and 6 GB of RAM. Android 10 is installed. The Android applications are compiled with Android Studio 4.0 with latest updates from 22th July 2020. The OpenSSL library has version 1.1.1.

For the visualization of the runtime measurements, we employ box plots since all schemes are based on random elements and we want to catch the statistical behavior. Additionally, we want to catch the variations induced by the usage of turbo clock and SMT. The colored boxes represent the values between the 25- and 75-percentile (Q_{25} and Q_{75}) meaning that 50% of the measurements lie in that range. The lines below and above the boxes represent the whiskers whose ends give the lowest and highest measurement point, respectively, which lies within $1.5\cdot$

$(Q_{75} - Q_{25})$ of the lower and upper quartile, respectively. The median Q_{50} is shown by the black horizontal lines. Outlying measurement points are drawn as circles.

For each box of the x64 diagrams we ran 15 iterations of our benchmarks. During this benchmark, 50 random messages of length 280 (maximum length for twitter messages) are created and 50 random key pairs are generated. Each message is signed 50 times with the same key and 50 times verified. For ARM, we perform 10 iterations with 25 keys and 25 sign/verify-operations for each.

The median is the employed metric when comparing the times. We consider runtimes as stable if the box plots are of small extent and no outliers far the median occur.

B. Evaluation of the runtime on Windows and Linux

Here, we compare the performance of qTESLA on two operating systems and with three compilers.

a) Performance comparison for C: First, we compare the performance of the three schemes when they are called within an executable without convenient interfaces for the users. We also want to investigate the influence of our parallelization on the performance compared to the baseline performance of the reference code. The results are summarized in Figure 1.

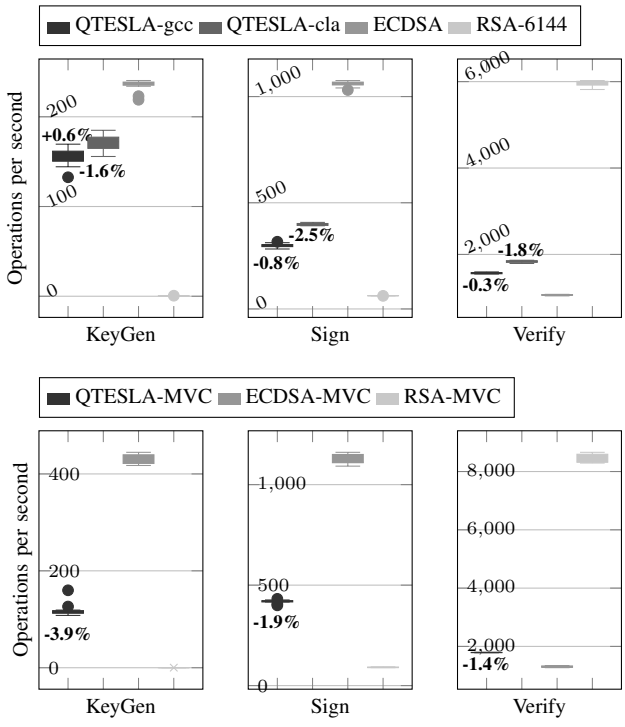

Fig. 1. Single-threaded runtime on Linux (top) and Windows (bottom) of C code. Difference to the reference implementation given in percent at the respective box.

The upper part of Figure 1 shows the runtime obtained on Linux. The key generation (KeyGen) shows that the Clang compiler (-cla suffix) results in slightly faster runtimes compared compiling with GCC (-gcc suffix). ECDSA, as expected, generates the most keys per second. qTESLA lies in the same order of magnitude and is about two times slower. RSA, on the other hand, is considerably slower. Its median for the key generation is $0.45\,ops/s$. For all schemes considered, the runtimes are relatively stable and just a few outliers with acceptable distance to the whiskers occur.

The situation in Windows is similar to Linux. However, ECDSA is about 25 % faster and qTESLA about 25 % slower. This makes ECDSA three times faster than qTESLA. The small box of RSA has median 0.2 and it is highlighted by a cross.

The middle part shows the signature generation (Sign). qTESLA and ECDSA again perform in the same order of magnitude with the Clang version being 2.5 times slower. RSA is 8 times slower than qTESLA and 14.5 times slower than ECDSA. All runtimes are stable. Under Windows the behavior is identical with the runtimes being the same for RSA, ECDSA and qTESLA compared to the Clang version.

Finally, the right side of Figure 1 shows the signature verification (Verify). In Linux, all schemes perform in the same order of magnitude, with RSA being three times faster than qTESLA and more than six times faster than ECDSA. In Windows, qTESLA and ECDSA have identical runtimes compared to the Linux equivalents, while RSA is 30 % faster.

In summary, we see that qTESLA is already competitive with the state-of-the-art signature schemes. Additionally, we see only a minor influence of compiler and platform. In general, Clang performs better than GCC and the Windows versions deliver comparable performance. The difference to the baseline reference version is very small. In most cases the re-arrangement of the code and the added pragmas even lead to slightly faster execution. Hence, we neglect the reference version in the following experiments.

b) Scaling behavior: Figure 2 summarizes the parallel scaling behavior of our new version.

In Linux, the KeyGen becomes about 1.7 times faster when employing two threads for both compilers. This increase is smaller for the MVC which is 1.4 times. When further increasing the number of threads, the performance increases slightly for Clang and MVC. GCC saturates for three threads. This behavior is not surprising since only five iterations of a loop are parallelized within KeyGen. When comparing the fastest configurations per OS, Linux (Clang, 4 threads) is 1.7 times faster than Windows (4 threads).

Considering Sign, Clang and GCC profit from employing three threads and increase their performance by a factor of 1.9 and 1.7, respectively. The MVC speeds up by a factor of 1.5 when employing two threads, has the same performance for three threads and becomes slower for four threads. In the case of sign, the fastest Linux configuration (Clang, 4 threads) is 1.1 times faster than Windows (MVC, 3 threads).

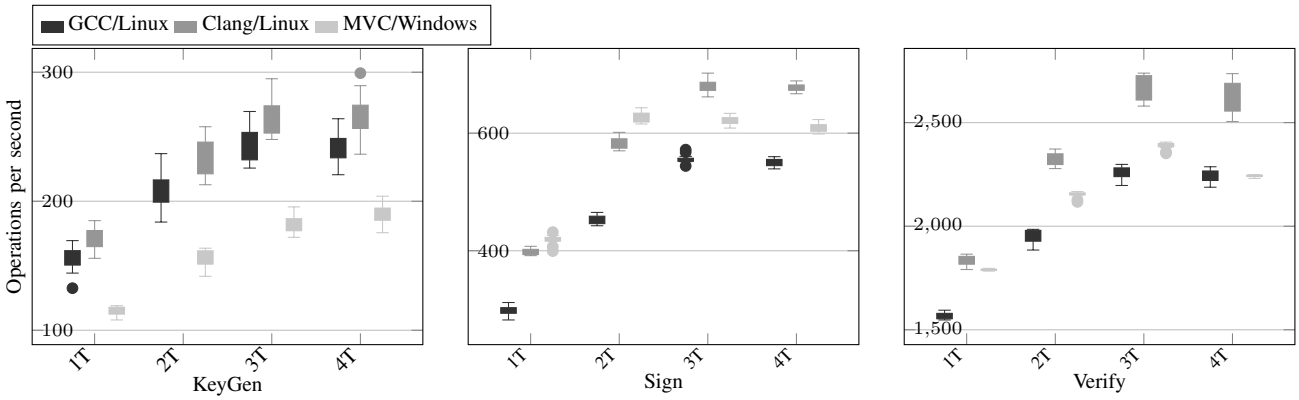

Fig. 2. Scaling behavior on different platforms. GCC/Clang are used under Linux, Microsoft Visual C under Windows.

For Verify, the compilers gain a speedup for three threads, with a factor of 1.5 for GCC, 1.4 for Clang, and 1.3 for MVC. Four threads do not improve the performance, as a loop with just five iterations is parallelized.

We sum up that employing parallelism with an adjusted number of threads considerably improves qTESLA's performance. The parallelism is only employed in appropriate parts, while the measurements consider the overall functions. In the respective parts, the parallel efficiency is higher and the additional consume of hardware resources is justified.

c) JCA/CSP evaluation: Here, we present the performance results when our optimized C implementation is wrapped behind the Java Cryptography Architecture (JCA). To that end, we choose the fastest C versions per operation system, i.e., Clang (4 threads) and MVC (3 threads) as reference. We also evaluate our Java version of qTESLA and employ the JCA-JRE implementations of RSA-6144 and ECDSA for comparison in Figure 3.

The most important point is that the performance of qTESLA is not significantly influenced by JNI and JCA. The worst case is Verify in Linux which is about 10 % slower than the straight C code. For all other cases, this deviation is smaller than 5 %. This demonstrates that the performance of the optimized C implementation is efficiently combined with the useability and flexibility of the JCA.

Comparing our C and our Java versions shows that for all three tasks the C version performs considerably better than the Java code. KeyGen in Linux is 6.5 times faster, while the factor is 2.6 in Windows. For Sign, C is faster by a factor of 4.1 and 1.8, respectively, while for Verify C is faster by a factor of 3.1 and 6.8. In general, the qTESLA Java code runs faster on Windows, in particular, for Sign which is nearly twice as fast.

For RSA and ECDSA, we see the opposite behavior since the runs under Linux deliver double the operations per second compared to Windows. Concerning the practicability, qTESLA, and in particular the C version, can compete with the standard schemes and offers the same interface, which allows quick adoption.

d) Optimized parameter set: Figure 4 shows the evaluation for the Clang version of qTESLA with optimized parameters derived in Section V.

KeyGen is considerably faster in the single-threaded case (1.9 times faster) as well as when employing 4 threads (double as fast), demonstrating the efficiency of the optimization.

For the Sign, we notice that the serial version becomes slightly slower. This is a consequence of increasing parameters L_E and L_S and is the expected behavior. However, we also see that the parallel version is about 10 % faster which practically validates the predictions drawn from Extra-P. In this case, we move more work into the parallelized code, i.e., the chance of being successful in a low number of iterations for the sampling loop decreases and a parallelization pays off earlier. By including the real runtime information, the modeler was able to automatically take this fact into account.

The single-threaded Verify does not change its performance at all. The parallelized version is slightly slower than the original version likely due to side effects of the parameter choice.

e) Results on Android: Finally, we also evaluated qTESLA on an ARMv8-A device. Since Android also has support for JNI, we can employ the same interface to the C code and compile the C part with Android Studio which internally employs Clang. ECDSA and RSA were again called from the OpenSSL library which is compiled as a static library with the Android Studio. The results are shown in Figure 5.

The C versions of all schemes in the upper part run considerably slower on the x64 computer than on the ARMv8-A device. We also see that qTESLA takes advantage of using 2 threads. For the Sign this nearly doubles the performance. The serial execution has the same speed as ECDSA for KeyGen and Sign, while the multi-threaded version is faster. For the Verify qTESLA is faster than ECDSA at all. RSA-6144 works very slow for all tasks.

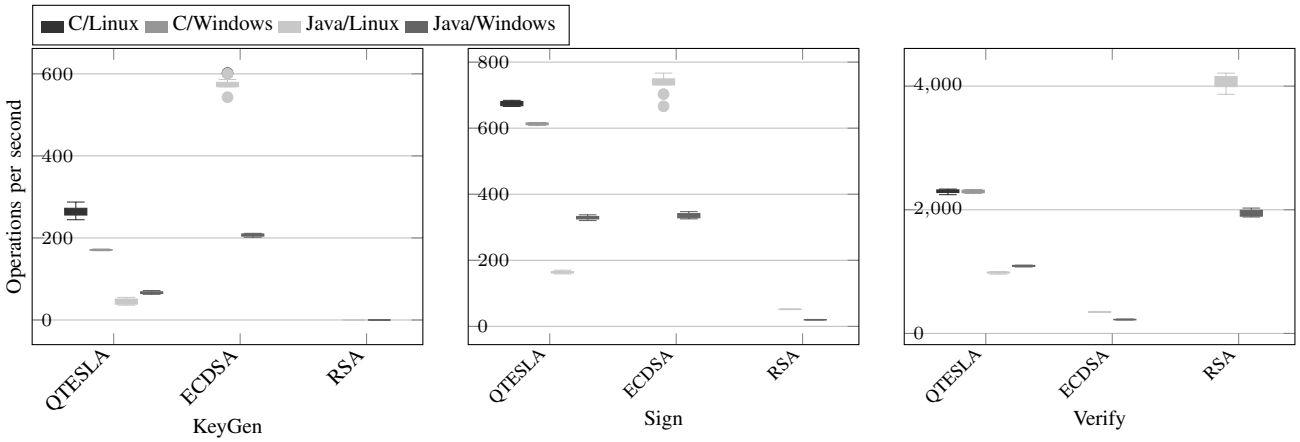

Fig. 3. Performance of JCA implementations of qTESLA in C and Java on Windows and Linux with the JCA versions of RSA and ECDSA.

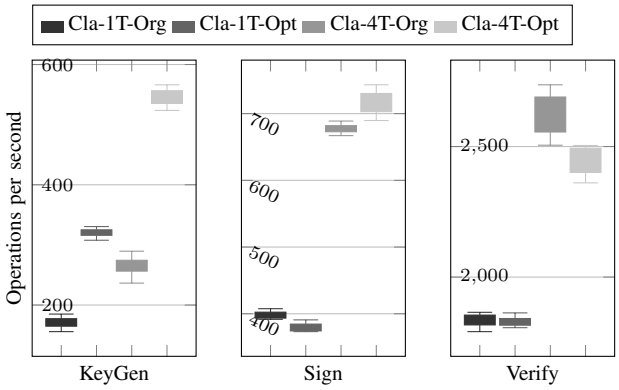

Fig. 4. Performance of original parallel and parameter optimized parallel version compiled with Clang.

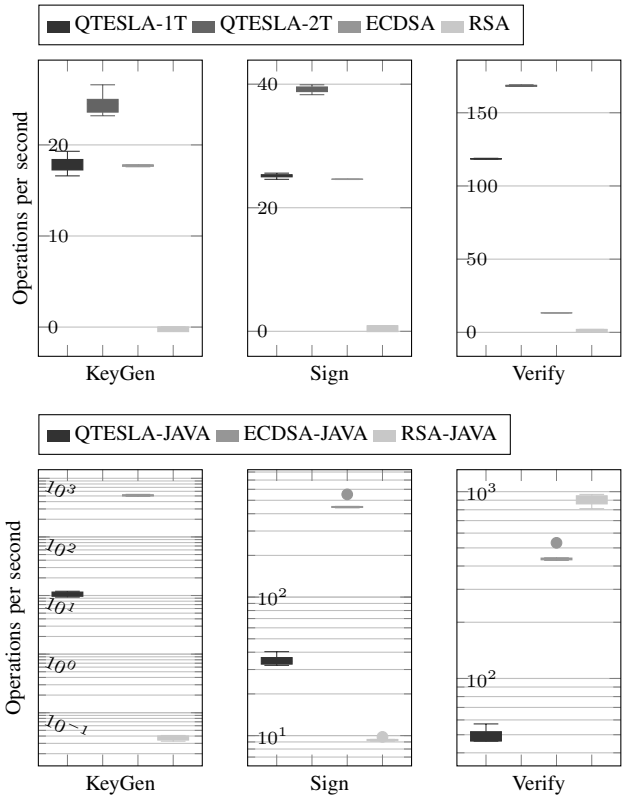

Fig. 5. Runtime comparison for C (top) and Java (bottom) on ARM.

Finally, the lower part of Figure 5 compares the performance of our Java implementation employing a logarithmic y-axis. For KeyGen, the Java qTESLA only executes half the operations per second than the C version (median 10.8 versus 24.5). However, for Sign the performance of the single-threaded qTESLA Java version is nearly at par with the parallel C implementation (median 34.4 versus 39.3). This indicates that the C compiler fails to create efficient code for the device. When verifying, the Java version is nearly four times slower than the parallel C code (median 46.9 versus 168.65). The very fast processing of ECDSA demonstrates that the functionality it depends on is efficiently realized in the Android runtime environment and future research must aim at optimizing qTESLA for ARM.

VII. Conclusion

We presented optimizations for the post-quantum signature scheme qTESLA. Regarding performance, this includes the parallelization of the three signature functions which are sped up despite their inherent serial parts and their overall short running times. Concerning usability and portability, we showed how to use qTESLA in Java behind the JNI and CSP interfaces. The performance on different architectures with different compilers and operating systems demonstrates that qTESLA competes with established signature schemes when they provide the same security level on classical hardware.

qTESLA implementations will be integrated into the CogniCrypt framework [21] to further increase its availability and usability. CogniCrypt is an intelligent Eclipse plugin developed within the DFG collaborative research center CROSSING at

TU Darmstadt. CogniCrypt provides a task-based code generation for developers on top of a state-of-the-art static analysis to identify cryptography misuses. Due to its significance it is an official Eclipse plug-in and is financially supported by various companies.

We will increase the serial performance of qTESLA by tackling the hotspot of sparse polynomial multiplication by replacing the implementation with more specialized algorithms to make qTESLA even more competitive.

ACKNOWLEDGMENT

Our work is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297. We thank A.-K. Wickert from TU Darmstadt for our cooperation to integrate qTESLA in CogniCrypt and N. Bindel from University of Waterloo for providing us the qTESLA-code with adaptable parameters.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] "Post-quantum cryptography pqc: Round 1 submissions," https://csrc.nist.gov/Projects/post-quantum-cryptography/Round-1-Submissions, accessed: 2021-02-23.

[3] E. Parliament, "Eu quantum manifesto: A new era of technology," http://qurope.eu/system/files/u7/93056_Quantum%20Manifesto_WEB.pdf, 2016.

[4] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography Standardization," https://csrc.nist.gov/projects/post-quantum-cryptography, Jan. 2017, accessed: 2018-07-23.

[5] E. Alkim, P. S. L. M. Barreto, N. Bindel, J. Krämer, P. Longa, and J. E. Ricardini, "The lattice-based digital signature scheme qTESLA," in *Applied Cryptography and Network Security*, M. Conti, J. Zhou, E. Casalicchio, and A. Spognardi, Eds. Cham: Springer International Publishing, 2020, pp. 441–460.

[6] N. Bindel, S. Akleylek, E. Alkim, P. S. L. M. Barreto, J. Buchmann, E. Eaton, G. Gutoski, J. Kramer, P. Longa, H. Polat, J. E. Ricardini, and G. Zanon, "qTESLA," https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/qTESLA-Round2.zip, National Institute of Standards and Technology, Tech. Rep., 2019.

[7] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, Nov. 2013. [Online]. Available: https://doi.org/10.1145/2535925

[8] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, , and D. Stehlé, "CRYSTALS–dilithium: Algorithm specification and supporting documentation," https://pq-crystals.org/dilithium/data/dilithium-submission-nist-round2.zip, 2019, submission to the NIST Post-Quantum Cryptography Standardization Project.

[9] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over NTRU," 2019, submission to the NIST Post-Quantum Cryptography Standardization Project.

[10] D. Micciancio and M. Walter, "Gaussian sampling over the integers: Efficient, generic, constant-time," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 455–485.

[11] R. K. Zhao, R. Steinfeld, and A. Sakzad, "Facct: Fast, compact, and constant-time discrete gaussian sampler over integers," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 126–137, 2020.

[12] R. Cole and R. Hariharan, "Verifying candidate matches in sparse and wildcard matching," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 592–601. [Online]. Available: https://doi.org/10.1145/509907.509992

[13] V. Nakos, "Nearly optimal sparse polynomial multiplication," *IEEE Transactions on Information Theory*, vol. 66, no. 11, pp. 7231–7236, 2020.

[14] "Provably-secure lattice-based digital signature scheme qtesla," https://github.com/qtesla/qTesla, accessed: 2021-01-11.

[15] T. L. of the Bouncy Castle, "Bouncy castle crypto apis," https://www.bouncycastle.org/, 1998.

[16] S. Liang, *The Java native interface: Programmer's guide and specification*. Addison-Wesley Professional, 1999.

[17] D. Kurzyniec and V. Sunderam, "Efficient cooperation between java and native codes – jni performance benchmark," in *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[18] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe api in the wild," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 695–710. [Online]. Available: https://doi.org/10.1145/2814270.2814313

[19] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. G. E. Wolf, "Score-P: A Unified Performance Measurement System for Petascale Applications," in *Competence in High Performance Computing 2010 : Proceedings of an International Conference on Competence in High Performance Computing, June 2010, Schloss Schwetzingen, Germany / ed. by Christian Bischof*, ser. SpringerLink: Springer e-Books. Berlin [u.a.]: Springer, 2012, pp. 85–97. [Online]. Available: https://publications.rwth-aachen.de/record/125894

[20] A. Calotoiu, D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf, "Fast multi-parameter performance modeling," in *Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan*. IEEE Computer Society, Sep. 2016, pp. 172–181.

[21] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic APIs," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.