Postquantum SSL/TLS for embedded systems

Yun-An Chang[†], Ming-Shing Chen[†‡], Jong-shian Wu[†], Bo-Yin Yang[‡],
[†]Department of Electrical Engineering, National Taiwan University, Taiwan
[‡]Institute of Information Science, Academia Sinica, Taiwan

Abstract—The "the internet of things" will require security infrastructure on small devices. This task is made more difficult as large quantum computers may appear soon and break currently standard PKCs (public-key cryptosystems). In anticipation, PKCs which can survive quantum computing ("postquantum cryptosystems", or PQCs) are actively being studied. However, effort put into building infrastructure for PQCs has been insufficient, in particular w.r.t. the lack a comprehensive library with a quantum-computing-resilient option for each public-key task. We present such a postquantum SSL/TLS library using publicly available parameters. We adapted this library from PolarSSL rather than the more popular OpenSSL because it was a much cleaner code base to work from. We have also refactored the original PolarSSL codebase to facilitate the incorporation of future cryptosystems. While testing is yet incomplete, both throughput and code size seem reasonable, facilitating adoption in resource-limited devices.

Index Terms—Post-Quantum, Cryptography, Embedded System, SSL, TLS

Extended Abstract

Work also supported by National Science Council, National Taiwan University and Intel Corporation under Grants NSC102-2911-I-002-001 and NTU103R7501.

1 Introduction

We human beings harbor an obvious need for security (authenticity, integrity, and secrecy). This is combined with (a) an insatiable appetite for (digital) data collection, storage and archiving, and (b) pervasive and ever-increasing widgets. Many information-security catastrophes are waiting to happen in modern society. We describe one effort to make things better by building better infrastructure. This let us phase PKCs in and out more easily and lessen the blow of catastrophic failure (as represented by quantum computing). We demonstrate empirically that the approach works for PCs and embedded systems.

1.1 TLS and public-key cryptography

The Transport Layer Security (TLS), as successor to the Secure Sockets Layer (SSL), is probably the most widespread cryptographic protocol for providing communication security over today's Internet. The first step of TLS protocol is the handshake which let two parties establish a shared secret for further use.

There are many Public Key Cryptography (PKC) techniques that may be involved during a handshake. One common mode includes a Diffie-Hellman (DH) key exchange (in which two parties compute a shared secret), and digital signatures which prevents man-in-middle attacks in the DH protocol. Central in this process is the Public Key Infrastructure (PKI) which provides trusted public key for both parities by pre-sharing a public key from a certificate authority (CA).

1.2 Why post-quantum public-key cryptography (PQC)

Popular PKC primitives like RSA and elliptic curve cryptography (ECC, mainly based on the hardness of the discrete log problem) both have the same vulnerability to quantum computers [1]. It is unclear when quantum computers will become widely available, but we can choose post-quantum (PQ) cryptography, PKC that is believed to be secure against quantum computers. Of course, PQCs are often PKCs of independent merit, but one important reason to pursue PQC now is to prevent "intercepted today, decrypted tomorrow".

We incorporated two cryptographic schemes into a standard TLS implementation for key exchange and digital signature and proposed some new ciphersuites for these schemes. The key exchange is based on a lattice-based scheme from Zhang et al [2] and its security depends on the hardness of finding short vectors in a lattice. The security of the digital signatures TTS and rainbow [3] depend mainly on the hardness of solving systems of multivariate equations and the MinRank problem.

1.3 What SSL library for an embedded system?

While SSL/TLS is now the de facto security standard in today's Internet, it also became a big and complex protocol which was specified by a dozens of RFCs after many updates. For incorporating new properties into SSL/TLS, it's natural to start with a well developed open-source implementation. Today's embedded systems, especially the sensors in today's "internet of things" has many resource restrictions including computation power, storage, and energy. As a result we have to wisely pick and carefully modulate

a library while providing a standard SSL/TLS to embedded systems.

Although the most popular implementation for SSL/TLS is OpenSSL [4], its codebase is also notorious for being an behemoth put together in ad-hoc fashion. We want to make embedded systems a fundamental concern instead of an afterthought so we started with PolarSSL [5], a more light-weighted implementation.

1.4 Related works and post-quantum TLS

J. W. Bos et al [6] implemented a Post-quantum key exchange for TLS with hardness based on the RLWE (ring learning with errors) problem. They replaced only the key exchange part of the handshake in OpenSSL, which is targeted to a general system. Other earlier research using PKC in M2M world may be found in [7] [8]. It is safe to say that no one has tried for a comprehensive, clean approach to updating SSL to be quantum-computing-resilient.

1.5 Our contributions

We present a full post-quantum TLS base on PolarSSL. Our modification contains a Latticed based key exchange, multivariate public key signature schemes, appropriate cipher suite additions, and has generic interfaces for future additions. All of these new features are implemented in a portable C manner which is suitable for low resourced devices as well as generic platforms.

2 Selection of Post-Quantum primitives and parameters

2.1 Key exchange protocol

There are many lattice-based key exchange schemes proposed recently [2], [6], [9]. We started with the scheme from Zhang et al. [2] and adapted the protocol for the use of PKI by sending the public key along the key exchange, and verifying the validity of the key later. The tweaked algorithm is shown in Fig. 1

Fig. 1. Diffie-Hellman-like lattice-based key exchange

Party i		Party j
Public Key: $p_i = as_i + 2e_i \in \mathbf{R}_q$		Public Key: $p_j = as_j + 2e_j \in \mathbf{R}_q$
Secret Key: $s_i \in \mathbf{R}_q$		Secret Key: $s_j \in \mathbf{R}_q$
where $s_i, e_i \leftarrow_r \chi_\alpha$		where $s_j, e_j \leftarrow_r \chi_\alpha$
$x_i = ar_i + 2f_i \in \mathbf{R}_q$		$y_j = ar_j + 2f_j \in \mathbf{R}_q$
where $r_i, f_i \leftarrow_r \chi_\beta$		where $r_j, f_j \leftarrow_r \chi_\beta$
	$\xrightarrow{x_i, p_i}$	
		$k_j = (p_i c + x_i)(s_j d + r_j) + 2g_j \in \mathbf{R}_q$
		where $g_j \leftarrow_r \chi_\beta$
		$w_j = \mathsf{Cha}(k_j) \in \{0,1\}^n$
	$\xleftarrow{y_j, w_j, p_j}$	
$k_i = (p_j d + y_j)(s_i c + r_i) + 2g_i$		
where $g_i \leftarrow_r \chi_\beta$		
$\sigma_i = \mathsf{Mod}_2(k_i, w_j) \in \{0,1\}^n$		$\sigma_j = \mathsf{Mod}_2(k_j, w_j) \in \{0,1\}^n$
$sk_i = \mathsf{H}_2(i, j, x_i, y_j, w_j, \sigma_i)$		$sk_j = \mathsf{H}_2(i, j, x_i, y_j, w_j, \sigma_j)$

$$c = \mathsf{H}_1(i, j, x_i) \in \mathbf{R}, d = \mathsf{H}_1(j, i, y_j, x_i) \in \mathbf{R}$$

TABLE 1
Selected primitives for signature

Scheme (over $\mathbb{F}_{31}$)	Security (bits)	Signature (Byte)	Digest (Byte)	Pubkey (Byte)	Seckey (Byte)
TTS					
(24,20,20)	80	40	24	53600	8608
(26,24,(2,4),24)	128	50	32	107900	13704
Rainbow					
(24,20,20)	80	40	24	53600	60960
(26,24,(2,4),24)	128	50	32	107900	112884

In the above protocol, $\mathbb{Z}$ denotes the ring of rational integers, $\mathbf{R}$ denotrs $\mathbb{Z}[x]/(x^n+1)$, $\mathbf{R}_q$ denotes $\mathbb{Z}_q[x]/(x^n+1)$, and $\leftarrow_r \chi_\alpha$ denotes a random choice from χ_α, which is a discrete Gaussian distribution centered at 0 with standard deviation α. The hash function H_1 also has a output space of χ_γ. The $\mathsf{Cha}()$ function checks every coefficient to see if it is in $\mathbb{Z}_{q/2}$, returns 0 if so and 1 otherwise. The function $\mathsf{Mod}_2(x, y) \mapsto \Big((x + y\dot{(}q-1)/2) \mod q\Big) \mod 2$ is also applied to every coefficient of the polynomial. We implemented from [2] Parameter Set III, where $n = 2048$, $\alpha = 3.397$, $\gamma = 161.131$, and $q = 3845762179574480897$ (designed for 128 bit security on par with AES128) and Parameter Set I where $n = 1024$, $\alpha = 3.397$, $\gamma = 101.919$, and $q = 1099511627689 = 2^{40} - 87$ (80 bits design security).

2.2 Signature schemes

Multivariate public key cryptography (MPKC) [10] is another one of typical PQCs. The public key of MPKC is a trapdoored multivariate quadratic polynomial map $\mathcal{P}$ which is (hopefully) hard to invert. Its general form is: $\mathcal{P} : \mathbf{w} = (w_1, \ldots, w_n) \mapsto \mathbf{z} = (z_1, \ldots, z_m)$, Where each $z_k := \sum_i P_{ik} w_i + \sum_i Q_{ik} w_i^2 + \sum_{i>j} R_{ijk} w_i w_j := p_k(\mathbf{w})$. Therefore evaluating a quadratic polynomial is the same as verifying a multivariate digital signature. The public map $\mathcal{P}$ decompose as $\mathbf{w} \overset{S}{\mapsto} \mathbf{x} = \mathrm{M}_S\mathbf{w} + \mathbf{c}_S \overset{\mathcal{Q}}{\mapsto} \mathbf{y} \overset{T}{\mapsto} \mathbf{z} = \mathrm{M}_T\mathbf{y} + \mathbf{c}_T \mapsto \mathbf{z}$, where S and T are unknown linear maps forming part of the private key, and the easily invertible quadratic map Q may contain parameters.

We choose Rainbow and TTS as MPKC signature schemes for authenticating the communications in the DH-like key exchange protocols. For 80-bit security, we use Rainbow/TTS over the finite field $\mathbb{F}_{31}$ with structure $(24, 20, 20)$, which corresponds to $m = 40$, $n = 64$. This was proposed in [11]. For higher design security, we propose a new structural parameters $(26, 24, (2, 4), 24)$, which has $m = 52, n = 80$. Details for selected primitives are listed in table 1.

2.2.1 Size of public key and other consideration

TTS has smaller secret keys compared to Rainbow and other MPKCs, saving storage resources. But MPKCs have large public keys which also is a drawback of

Rainbow/TTS in embedded systems. We can choose like Petzoldt et al [12] with "circulant" variants of MPKC with smaller public key. In such variants, Pubkey can be reduced to about 10 kbytes with similar parameters. We might also invest in a caching mechanism of public keys on the server side in a local network, this would also significantly reduce the issue to an overhead of only the size of the digest(hash) of the public key. This would work best in a sub-network containing only small and static members, but a new or un-cached public key might cause a cache miss and request a re-start of handshake protocol. There is also an extra communication which is not in current TLS.

2.2.2 Current Implementations

TTS is suitable for microcontroller, ASIP, or normal windows computer implementation [11], [13], [14]. We are using here $\mathbb{F}_{31}$ so it is important to accumulate computations to postpone reduction modular 31. [11] does this as well as using Wiedemann's algorithm for solving linear equations in signing process, which makes the computational data flow less dependent on the data (reducing the probability of side channel attacks) and makes full use of SIMD at the same time. As shown in [13], arithmetic over small fields on microcontrollers today can be implemented well using lookup tables, as well as SIMD instructions.

2.3 Post-quantum cipher suites for extending current TLS

Since many constants had been specified for identifying crypto elements in current TLS standard, we had to add new identifying constants for extending TLS to the chosen post-quantum primitives. These new suites are proposed for our post-quantum TLS: `LATTICEE-TTS-WITH-AES128-GCM-SHA256` `LATTICEE-RIANBOW-WITH-AES128-GCM-SHA256` `LATTICEE-RSA-WITH-AES128-GCM-SHA256` `LATTICEE-ECDSA-WITH-AES128-GCM-SHA256` . We choose these new constants and perform the protocol by following current TLS standard, but we have to mention these new constants can only be recognized with our modified PolarSSL before becoming a standard.

3 Hacking PolarSSL

3.1 Introduction to PolarSSL

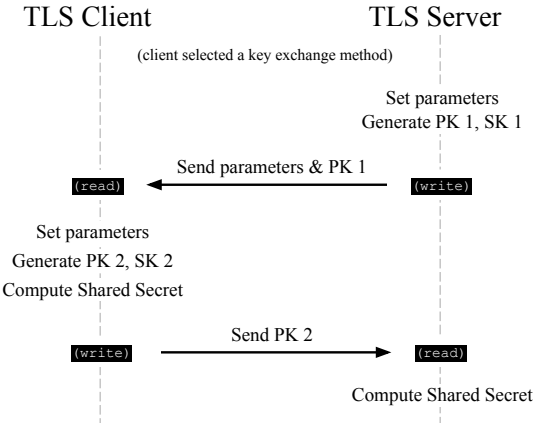
PolarSSL is an open-source project to implement SSL/TLS in a clean and modularized way to make components less tightly coupled and more understandable. The top module directly depends on some other components, including a TCP/IP module, a RNG module, a Cipher module, a Public Key module, a Hashing module, and an X.509 module. PolarSSL has unified "generic crypto interfaces" for several primitives but are lacking others, including a Diffie-Hellman-like key exchange protocol. Because PolarSSL today implements DH and ECDH separately with some nontrivial differences, there is redundancy in the codebase: Many almost-identical code snippets related to Diffie-Hellman key exchange occur twice for DH and ECDH. The ECDH is further restricted to Weierstrass form, which has more potential vulnerabilities.

In contrast to OpenSSL which containing tremendous assembly code for various hardware and crypto primitives, most code in PolarSSL are in the portable C manner. PolarSSL is also a self-contained library which depended only on stand C library now. It's hard to judge this self-contained property. Although it diminished the necessity for extra library which is good for small storage devices and increased the security from compromised outer libraries, a well-tuned big-number library did improve the performance for cryptographic primitives, see sec. 4.2.2. For targeting low resource devices in this work, We choose to follow the self-contained and portable properties in PolarSSL. Our modification started from PolarSSL version 1.3.8 (git version: `1910aa`).

3.2 A new Diffie-Hellman interface for PolarSSL

In order to add new key exchange mechanisms into SSL, a unified interface for Diffie-Hellman-like protocols would be helpful, since almost no change in the SSL part are needed to add a new key exchange protocol when the implementation details are hidden by the interface as a black box.

Fig. 2. A Typical DHE Workflow



A typical Diffie-Hellman-like workflow running in ephemeral mode will have the pattern as shown in Fig. 2. Basically there are only two computational parts, one to generate a public-secret key pair, the other to compute a shared secret after receiving the peer's public key. Because the actual data sent over the wire with DH or ECDH is different, we also need

to carefully extract required Diffie-Hellman parameters, key pairs, and computation results from and to the internal state of the SSL module. Therefore our proposed interface for Diffie-Hellman-like key exchange is responsible for the computations and I/O utilities that are meant to be used directly by the top SSL module. In the future, we may abstract further. Our generic DHE currently performs these actions:

1) `set_params` : To specify the key exchange parameters.
2) `gen_public` : To compute a public key pair.
3) `compute_shared` : To compute a shared secret.
4) `{write,read}_ske_params` : To process the server-to-client ServerKeyExchange handshake message.
5) `{write,read}_public` : To process the client-to-server ClientKeyExchange handshake data.
6) `write_premaster` : To process the resulting shared secret of a complete key exchange.
7) `read_from_{self,peer}_pk_ctx` : Extract parameters and server's public key from an X.509 certificate.

3.3 Using our generic DH in PolarSSL

With the new Diffie-Hellman(DH) interface, we can focus on the computation functions for a new DH-like schemes without worrying about SSL/TLS interaction. With all DH-like primitives in a unified interface, it remains to clean the original PolarSSL code to be consistent with the new DH interface. The original PolarSSL key exchange primitives (DHM and ECDH) is thus re-written with our new DH interface.

In the original PolarSSL implementation, the code size was targetted using C macro preprocessors to control the enabling or disabling of primitive hardcored into program flows. This feature resulted in larger source code size in each function, redundant run-time check to find the enabled options, and most made the source code harder to read from a software design viewpoint. Our approach unified the program flow by using our abstracted DH interface in the SSL-layer functionality. This makes the code smaller and easier to read and understand. The results shows that our approach has benefits in software engineering without corresponding cost in te code size.

4 Experiments and Benchmarks

4.1 Regression Testing

At the moment the refactored PolarSSL structure passes through black-box regression tests in 88 ciphersuites dealing with Diffie-Hellman (as HTTPS server and client) and signatures. We did not test any non-DH key exchange (i.e., using a public-key encryption method). We did not focus on non-DH PKC as a key exchange method because it would expose the "break once decrypt everything before" problem.

4.2 Benchmarking

Although post-quantum security is our first concern for extending SSL/TLS implementation, we reported some benchmarking results of our implementation to show it's applicable to small devices.

4.2.1 Code size

We measured the modified code size of static library in X86_64 and ARM cortex platforms. The code size is below 1MByte with all features on in both platforms and can be further reduced by removing unnecessary crypto elements. The binary size with our DH interface is slightly larger than that of the original because we added a new wrapper for adapting original ECDH and DHM functions to our new interface. The memory footprint of the binary with our new PQ features is below 1 MBytes and runtime memory requirement for a handshake is about 128kByte (measured by `valgrind` [15]). This low requirement of storage/memory resources are advantages inherited from PolarSSL and easily satisfiable for today's low resource devices.

4.2.2 Benchmarking cryptographic primitives

We show the performance for DH and signature primitives in table 2. The experiment was performed in an Intel CPU Xeon E3-1245v3(3.40GHz) machine. ECDHE, RSA, and ECDSA are original PolarSSL implementations. Lattice exchange, TTS, and Rainbow are our new post-quantum primitives for extending TLS.

In the lattice exchange, we tried to change the big-number library to GNU big-number arithmetic library(GMP) [16] and resulted a performance improvement in order of magnitude from original PolarSSL big-number library. We decided to maintain the self-contained and portable C properties for minimizing the resource requirement at latter experiments. These are many big-number arithmetics in RSA and EC computations, and the original PolarSSL implementation could also be improved with a well-tuned arithmetic library.

4.2.3 Throughput for handshake protocols

The performance of full TLS handshake is reported in table 3. These data was measured in an Intel CPU Xeon E3-1245v3(3.40GHz) machine. The other parts of cipher-suite are AES128-GCM for authentication encryption and SHA-256 for hash function. We slightly modified the example HTTPS server and client which finished a connection right after a successful handshake in PolarSSL for benchmarking handshake only. The PolarSSL HTTPS server running in the background listening on a port locally for TLS connection, and we ran 100 client program sequentially in the same machine to diminish the effect of networking.

TABLE 2
Performance of cryptographic primitives in PolarSSL

Computation	Throughput
ECDHE (secp521r1) exchange	189 exchange/s
RSA (2048-bit) sign	482 sign/s
ECDSA (secp256r1) sign	1629 sign/s
RSA (2048-bit) verify	15883 verify/s
ECDSA (secp256r1) verify	434 verify/s
Lattice exchange [2, I]	9.2 exchange/s
Lattice exchange (using GMP) [2, I]	62.5 exchange/s
Lattice exchange [2, III]	4.6 exchange/s
Lattice exchange(using GMP) [2, III]	28.5 exchange/s
TTS (80b) sign	24666 sign/s
TTS (128b) sign	15381 sign/s
Rainbow (80b) sign	4056 sign/s
Rainbow (128b) sign	2199 sign/s
TTS/Rainbow (80b) verify	12302 verify/s
TTS/Rainbow (128b) verify	6136 verify/s

The ECDHE-RSA and ECDHE-ECDSA are original PolarSSL implementation and tested with example certificates in the PolarSSL. Comparing to table 2, their throughput in table 3 are lower because of computations in SSL/TLS layer. For experiments with TTS, the TTS public key was communicated with a self-signed certificate which is pre-trusted by both sides. We can see the throughput of signature algorithms from table 2 are high above the the throughput of handshake protocol which implying no significant effect from signature algorithms. The full handshake throughput of lattice key exchange is down to the same with only lattice key exchange might be caused from our current slower implementation in key exchange than other parts in handshake of PolarSSL.

TABLE 3
Performance of full TLS handshake in PolarSSL

Cipher suite	Throughput handshakes/sec
ECDHE(secp521r1)-RSA(2048-bit)	20.0
ECDHE(secp521r1)-ECDSA(secp256r1)	18.1
ECDHE(secp521r1)-TTS(128b)	18.2
LATTICE(III)-TTS(128b)	4.5

5 Conclusion and Future Works

Making libraries extensible and clean is clearly a good idea. OpenSSL is probably too laden with legacy code to be salvaged this way but we have refactored most of PolarSSL in this fashion and have successfully passed regression tests on our new DH interface. Note also that PolarSSL already have a generic PK encrypt interface so there is no reason why the GPL'ed libraries from NTRU [17], a PQ encryption scheme, cannot be inserted instead of RSA as a key exchange method if we are not concerned about forward secrecy (which we are). One important refactoring lacking now is probably an authenticated cipher interface for new advanced authenticated encryption modes [18].

Work still remains to provide faster (and tested correct) code for the primitives we choose and optimize them to the best of our ability for common platforms.

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, Oct. 1997.

[2] J. Zhang, Z. Zhang, J. Ding, and M. Snook, "Authenticated key exchange from ideal lattices," Cryptology ePrint Archive, Report 2014/589, 2014, http://eprint.iacr.org/.

[3] A. I.-T. Chen, C.-H. O. Chen, M.-S. Chen, C.-M. Cheng, and B.-Y. Yang, "Practical-sized instances of multivariate pkcs: Rainbow, tts, and lic-derivatives," in *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, 2008, pp. 95–108.

[4] "Openssl," https://www.openssl.org/.

[5] "Polarssl," https://polarssl.org/.

[6] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the tls protocol from the ring learning with errors problem," Cryptology ePrint Archive, Report 2014/599, 2014.

[7] J.-R. Shih, Y. Hu, M.-C. Hsiao, M.-S. Chen, W.-C. Shen, B.-Y. Yang, A.-Y. Wu, and C.-M. Cheng, "Securing M2M with post-quantum public-key cryptography," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 3, no. 1, pp. 106–116, 2013.

[8] D. J. Malan, M. Welsh, and M. D. Smith, "Implementing public-key infrastructure for sensor networks," *ACM Trans. Sen. Netw.*, vol. 4, no. 4, pp. 22:1–22:23, Sep. 2008.

[9] C. Peikert, "Lattice cryptography for the internet." *IACR Cryptology ePrint Archive*, vol. 2014, p. 70, 2014.

[10] J. Ding and B.-Y. Yang, "Multivariate public key cryptography," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Springer-Verlag, 2009, pp. 193–241.

[11] A. I.-T. Chen, M.-S. Chen, T.-R. Chen, C.-M. Cheng, J. Ding, E. L.-H. Kuo, F. Y.-S. Lee, and B.-Y. Yang, "SSE implementation of multivariate PKCs on modern x86 CPUs," in *CHES 2009*, Lausanne, Switzerland, September 2009, pp. 33–48.

[12] A. Petzoldt, S. Bulygin, and J. Buchmann, "Cyclicrainbow - A multivariate signature scheme with a partially cyclic public key," in *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, 2010, pp. 33–48.

[13] B.-Y. Yang, J.-M. Chen, and Y.-H. Chen, "TTS: High-speed signatures on a low-cost smart card," in *Cryptographic Hardware and Embedded Systems — CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, ser. Lecture Notes in Computer Science, vol. 3156. Springer, 2004, pp. 371–385.

[14] B.-Y. Yang, C.-M. Cheng, B.-R. Chen, and J.-M. Chen, "Implementing minimized multivariate PKC on low-resource embedded systems," in *Security in Pervasive Computing, Third International Conference, SPC 2006, York, UK, April 18-21, 2006, Proceedings*, 2006, pp. 73–88.

[15] "Valgrind-3.10.0," http://valgrind.org/.

[16] "Gmp," https://gmplib.org/.

[17] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, 1998, pp. 267–288.

[18] "CAESAR: Competition for authenticated encryption: Security, applicability, and robustness," http://competitions.cr.yp.to/caesar.html, a competition partially sponsored by NIST.