White Paper

Efficient Implementation of Lattice Cryptography

Introduction

Are Utimaco products ready for post-quantum safe asymmetric cryptography today? After the last few months working on my bachelor's thesis (by Patrick Mertens), I can confirm that porting a brand-new algorithm to Utimaco Hardware Security Modules (HSMs) and optimizing it using profiling techniques has been a pleasant, straightforward task. So indeed, Utimaco supplies an ideal ecosystem for current research and advanced cryptography. This article summarizes my efforts and aims to validate this proposition.

As an entry point for all digital communication, the initial key exchange (KEX) is crucial for ensuring channel security. After this process has been completed, all parties share a joint secret which is used to derive the key for a symmetric encryption algorithm such as AES. Hence, if someone is able to break the cryptography behind this initial key agreement, the common secret can be obtained. Subsequently, the eavesdropper can decrypt all messages using this key.

Most of the current KEX algorithms, like Diffie-Hellman, depend on the fact that prime factorization is a hard problem and computationally expensive for modern hardware. However, a quantum computer can factorize primes easily in polynomial time using Shor's Algorithm. With recent advancements in quantum research, a successful deployment of such a machine could happen earlier than initially thought possible. What's more, an attacker could decrypt past communication, secure at the time, with this tool if all communication is saved until that point.

Luckily, work has been done in the area of lattice-based cryptography and protocols promising post-quantum security are already published. One such proposal is the NewHope algorithm by E. Alkim, L. Ducas, T. Pöppelmann and P. Schwabe, which offers various performance optimizations of an earlier scheme conceived by C. Peikert. Both base their cryptographic security on the ring learning with errors (Ring-LWE) problem, which was shown to be difficult and hence unbreakable for both quantum and conventional computers by O. Regev in 2005. NewHope has gained popularity among the security community, with Google already testing it in a fraction of the connection setup between the Chrome Browser and their servers.

Utimaco HSMs form the essential backbone to many security setups, providing secure storage and computing devices for key management. The CryptoServer (CS) HSMs come as plug-in PCIe cards and have strong physical protection against intruders built into the hardware, as well as the software system. The cards feature extensive tamper resistance, evidence and detection abilities, in order to delete stored keys and data immediately upon being attacked. The CryptoServer software provides a secure messaging channel for communication between a host PC and the CS.

This secure channel is implemented using today's best practices and certified according to current standards. However, advances in quantum computing may weaken forward security at some point in the future. To counteract, Utimaco customers can implement a hybrid approach to a post-quantum-safe secure channel already today, using the CryptoServer SDK provided by Utimaco. This, and the fact that its CPU is a digital signal processor (DSP) from Texas Instruments (TI), makes implementing NewHope an important case study for post-quantum cryptography.

In my bachelor's thesis, titled "Efficient Implementation of Lattice Cryptography", I am porting the public domain reference implementation of NewHope to run on Utimaco's CryptoServer CSe. This is done in collaboration with Prof. G. Ascheid from the Institute for Communication Technologies and Embedded Systems (ICE) of RWTH Aachen. After integrating NewHope into a custom firmware module, a deep analysis and improvements of its performance on the DSP are carried out. Lastly, I propose further areas for optimizing run time, classified into three categories: general-purpose optimization of the firmware, adjustments specific to the processor used, and hardware-focused optimizations that may require either a different processor architecture or the use of a crypto co-processor.

The NewHope KEX algorithm

The following explanation is extremely simplified to keep it brief, please consult the NewHope paper for the full details. All calculations throughout the algorithm are done in a polynomial ring modulo parameter q, hence the name Ring-LWE. The authors of NewHope split the key exchange scheme into three main functions. During the key generation invoked by Alice (entity A), a seed is obtained, and the public polynomial a is generated from it. After sampling a secret s and a noise polynomial e, the calculated value b = a * s + e is sent to entity B along with the seed for a. In the subsequent function, named shared B, Bob executes a similar sampling of secret s' and error polynomials e' and e'', computing its shared polynomial v = b * s' + e''. Along with information helping with error recovery, the polynomial u = a * s' + e' is returned to Alice. In the final function, called shared A, the first entity multiplies the joint polynomial u * s. This leads both entities to the same polynomial as private key, after noise values have been dissolved by error reconciliation. To discard any excess information, the shared polynomials finally get hashed with SHA-3 to form the shared key for both entities.

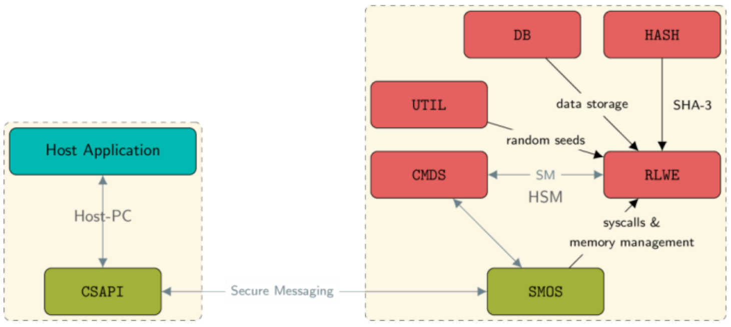
A notable enhancement to this process, introduced by NewHope, is the calculation of all polynomial multiplications using the number-theoretic transform (NTT), the specialization of the discrete Fourier transform (DFT) on polynomial rings.

To stress the system's true random number generator as little as possible, the algorithm relies on D. Bernstein's ChaCha20 stream cipher applied to a small seed for sampling all noise values.

Implementation & Adaptions

Porting NewHope's C reference implementation, which targets Intel's x86 architecture, to the HSM has been a clear-cut exercise. Most of the code does not depend on any system libraries and is therefore truly portable, as promised. Only the part to obtain the random seed is (obviously) dependent on the operating system, but Utimaco's utility module offers an easy way to obtain those on a CryptoServer.

To integrate the algorithm into the ecosystem, one has to design the public application programming interface (API) to be accessible to other modules on the CryptoServer and the external interface for usage by host applications. My RLWE module's public API consists only of NewHope's three functions for the KEX. The external interface builds upon the public API and stores the secret after the key generation, until the corresponding shared A function, in a secure, volatile database in the DB module. Remarkably, the design of the external API simply returns the common secret agreed upon by the KEX process. Ideally, the secret stays on the HSM or is, in practice, immediately consumed to generate a symmetric encryption key. This definitely requires tying in additional modules and is best done in conjunction with a dedicated cryptographic API like PKCS #11. It is omitted in my thesis, as it is largely boilerplate integration and unnecessary, since my work is meant as a demo rather than a fully integrated system. But future work can begin here smoothly and add a post-quantum safe layer on top of Utimaco's secure messaging system.



As the NewHope scheme computes a SHA-3 hash on the shared polynomials calculated in the last step, my implementation falls back to Utimaco's HASH module for that. All of these interactions with the firmware are visualized in the diagram and need to be declared as startup dependencies in the module link table.

Tests to verify the functionality are present in the small host application, accessing only the external interface. Since NewHope's reference implementation already comes with unit tests for the smaller functions on polynomials, I omitted those and focused on integration tests for the whole key exchange with a host as second entity.

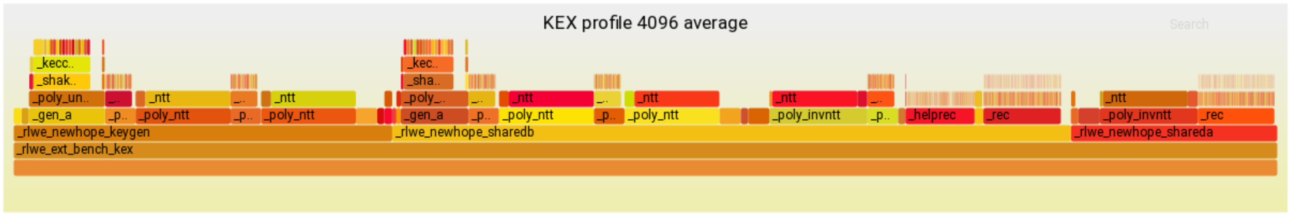
Performance Analysis

With the basic implementation of the RLWE module working and a successful key exchange confirmed by tests, my work is not over yet. Since the key exchange is such a critical part of communication, it is crucial to make it run as fast as possible. But, in order to iteratively enhance the performance of the KEX, a stable benchmark must first be established. My benchmark concept devotes a separate benchmark to each of the three public NewHope functions and sets up an additional one targeting the whole KEX. Each is invoked as an external function of the module, gets the number of desired runs as a parameter and returns the average cycle count taken. The first benchmark run produced a mediocre result of slightly less than 2.5 million CPU cycles for the whole KEX on average.

In order to get a deeper understanding of the run time of the computations, I then attached a profiler to the module. As the module is running on the HSM without possible external access to the CPU and RAM, none of the usual profilers like gprof would work here. Instead, a profiler based on hook functions fits in nicely. TI's compiler allows you to insert a custom hook call at the start and end of each function, which gets the address of the original function as a parameter. The hook saves the address, along with the current CPU timestamp counter, in a large buffer in memory, maintaining a count of the total number of entries.

Retrieval of the profiling data is performed through a supplementary external function, sending the contents of the profile buffer securely through the answer buffer. An initial problem that appeared here is that the size of the answer buffer is limited to 256 KiB by the secure messaging protocol, operating in between the host application and the CryptoServer. Therefore, the host has to read the buffer contents in multiple smaller calls of the interface. A mapping of the raw function addresses to their names is done by reading the address of the known start function and consulting the linker address map to calculate the load offset of the module in RAM. After this, the linear data set of function call and exit points (in cycle counts) is transformed into a chronological sequence of full call stacks and their durations, with a script written in Perl. Its output can be directly piped into the flame graph library from B. Gregg, producing an interactive graph visualizing stack changes over time.

To prevent obtaining different graphs for the same code version due to variations in execution flow and cache misses, the script was modified to take an average from multiple runs of the same targeted function. This is achieved by scanning the array of saved call stacks, to figure out where the next stack frame should be accumulated on to. The result of the initial visualization of 4096 runs can be observed below. The availability of such a profiling option on the HSM permitted a swift performance analysis, as described in the next section.



Iterative Optimizations

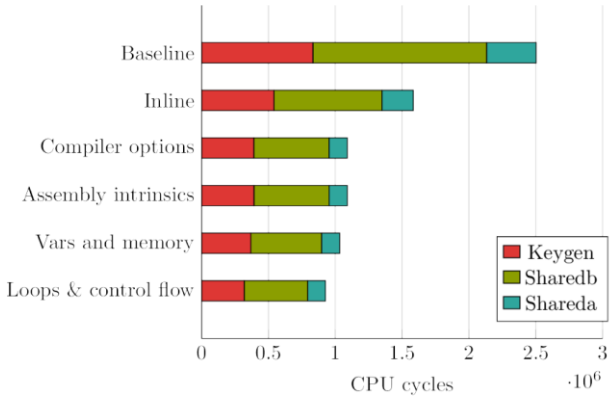
With help of the flame graphs and benchmark numbers, I could quickly determine bottlenecks and accelerate the program with changes, step by step. The following modifications proved to have a largely positive impact on performance.

First, I noticed the large amount of short function calls in the graphs, visible as tiny lines at the top. By inlining those functions, I was able to get rid of the call overhead and therefore accomplished a massive improvement with more aggressive compiler options for optimization. Ensuring that no two parameters of a function point to the same memory location (-mt) along with level 3 optimizations (-O3) yielded a 56.2% reduction in KEX run time.

Next, the very long instruction word (VLIW) DSP provides instructions that are more complex than can be expressed in a simple C statement, so the compiler cannot always recognize those patterns easily and then select the best machine code. A little help can be applied with assembly intrinsics which directly map to the right instruction. Evidently, these low-level changes produce hardly measurable results, but I was able to remove an array containing bit-wise inverses entirely by replacing it with the _bitr() intrinsic.

Keeping in mind the details of the processor architecture was useful as well. In various places throughout the algorithm, counter and index variables are instantiated as 64-bit integers. The DSP's smaller 32-bit registers makes calculations with these integers use two registers each, and thus they are highly ineffective. A shortening of these variables, wherever possible, enabled single instruction multiple data (SIMD) vectorization in one step.

Furthermore, there are already a lot of adjustments to NewHope, even though the implementation is declared to be a cross-platform reference. Some of these only really make sense regarding the widely used x86 architecture. For instance, the functions for both the ChaCha20 stream cipher and the Keccak algorithm had a manual loop unroll of factor 2 applied in their main loops. This duplication of code may provide better register utilization on Intel's processors, but the HSM's CPU is capable of an even better technique called loop pipelining, which effectuates maximal parallelization. Even without it, the TI compiler is able to automatically unroll loops, if greater hardware efficiency can be attained. I could easily activate the loop pipelining optimization by undoing the manual loop unroll for ChaCha20. Reverting the unroll in the larger Keccak loop led to faster computation as well, since the registers were already exhausted by a single iteration and the compiler had to temporarily store values in memory.



The diagram on the right summarizes the performance gains accomplished during this iterative analysis for the three functions of the key exchange.

Areas for Improvement

The optimizations carried out so far undoubtedly do not mark the end of performance growth. With a little more effort, the following areas deserve extended research and may provide useful insights. In general, the main focus for modification here should be on the expensive NTT computations and thereafter, the sampling of random noise polynomials, as those are the biggest bottlenecks.

For more general-purpose firmware optimizations, the NewHope authors themselves proposed the idea of caching the public polynomial a for multiple KEX requests. This can be easily incorporated into the API, but it certainly weakens the security of the protocol for a caching attack similar to Logjam. Moreover, NewHope's reliance on common cryptographic algorithms for hashing and random numbers means that it hugely profits from an efficient implementation of those.

Diving deeper into adjustments specific for the DSP could decrease run time costs further, especially as the NTT can be adapted to work better with the instruction pipeline, allowing more parallel computations. Its complex variant, the fast Fourier transform (FFT), is even included in TI's DSPLIB, promising hand-optimized assembly code.

Finally, the biggest capacity for tuning is possible through altering the hardware setup. A dedicated cryptographic co-processor could be designed to outsource tasks such as NTT computations. However, the effort seems unreasonable or even overkill, in comparison with the less radical software changes that are still viable.

Conclusion

To wrap up, I achieved a working port of the lattice-based NewHope algorithm to the Utimaco HSM and integrated it into the ecosystem. Benchmarks and profiling were developed to verify and give insights into modifications targeting maximum performance. The iterative changes resulted in a considerable improvement to the key exchange process, with additional suggested changes promising more optimization. Utimaco products embodied a supportive foundation for the implementation and the optimization workflow, enabling a completely secure key exchange for the future, usable today. This marks an important first step towards post-quantum cryptography on an HSM and opens the door for prospective work. If you are interested in current developments, join our dialogue about post-quantum cryptography!

About Utimaco

Utimaco is a leading manufacturer of HSMs that provide the Root of Trust to all industries, from financial services and payment to the automotive industry, cloud services to the public sector. We keep cryptographic keys and digital identities safe to protect critical digital infrastructures and high value data assets. Our products enable innovations and support the creation of new business by helping to secure critical business data and transactions.

Founded in 1983, Utimaco HSMs today are deployed across more than 80 countries in more than 1,000 installations. Utimaco employs a total of 170 people, with sales offices in Germany, the US, the UK and Singapore. For more information, visit https://hsm.utimaco.com/

Download the Utimaco Software HSM Simulator to get started immediately learning about HSM devices. It is a FREE fully functioning Software version of the Hardware HSM. The download package includes documentation on our product. Register here for your download.

Ready to take off?
Download our HSM simulator!
Register for free
Take me there

The Simulator download includes tools for creating user accounts, sample code and libraries for PKCS#11 Microsoft CNG, Java JCE and the Utimaco CXI API to link and test your code. The Simulator will run on a Windows or Linux platform. This is your opportunity to try out the Utimaco HSM technology easily and without initial cost.