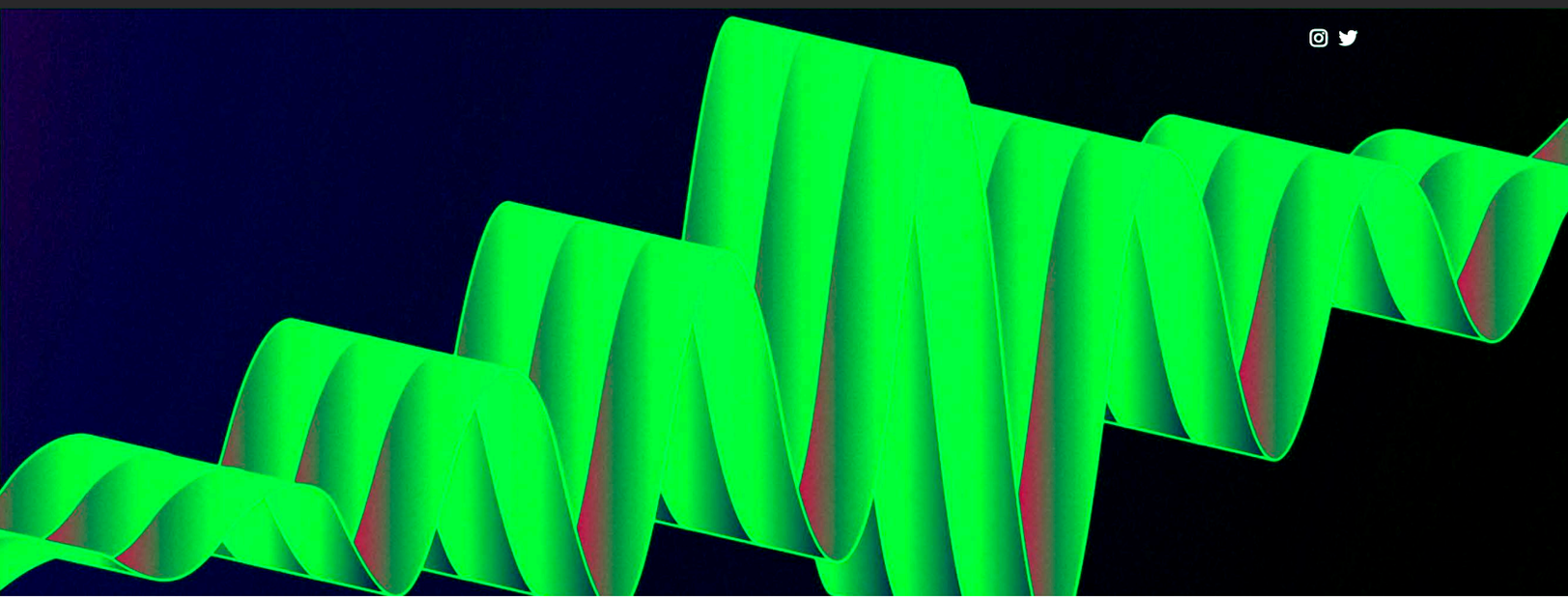


Optical Computing for Post-Quantum Cryptography

An Introduction

[*18 minute read*]

These days, if you do any reading on the subject of information security, you don't need to look very far to find a discussion of the problem of post-quantum security. It's a well-known issue with the infrastructure of the modern internet; the cryptographic tools that keep information in transit secure from prying eyes and malicious activity are imminently at risk from an entirely new method of computing. The culprit, quantum computing, enables the use of new calculation techniques that can easily solve some of the fundamentally difficult mathematical problems on which these cryptographic tools are based.

There have been two major avenues of research into a solution to this problem. The first is to fight quantum mechanics with quantum mechanics, creating cryptographic schemes (such as E91) that explicitly use quantum properties, typically entanglement, as a means of securing information. However, this approach requires dedicated quantum hardware.

The second approach is to find an alternative mathematical problems or features which are not known to be solvable even with a quantum computer. This is easier said than done; finding suitable mathematical problems for cryptography is not an easy task even without having to verify that they are also secure against quantum attacks, but there has been some exceptional progress in the field.

Amongst these alternatives is a very promising category of techniques collectively known as *lattice-based* cryptography. The fundamental mathematical problems that secure lattice based cryptography are extremely flexible; not only do they enable replacements for vulnerable cryptographic schemes, but they can be used to construct new and useful models of cryptography that allow us to do things that we couldn't do before.

In future articles, we'll be looking at how our optical computing technologycan be used to accelerate key operations in cryptographic schemes based on these lattice methods. For now, in this article, we take a look at *why* we need alternatives. This isn't an in-depth explanation of all cryptography, or how quantum computers work; rather, the aim is to provide an accessible understanding of the nature of the quantum threat to existing techniques. We'll start with the (very) basics of how we can securely send messages, and go from there.

Cryptography: The (Very) Basics

Cryptography is the transformation of information for the purposes of security. This can take many different forms for many different applications, but for now we're going to focus on methods which are used to obscure information from all but those authorised to see it.

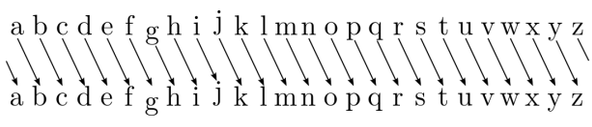
Let's take a classical example. We want to send a message to a friend, but we don't want anybody else to be able to read it.

We can't send this information in a readable form (known in cryptography as "plaintext"), so we want to transform the message into an encrypted *ciphertext*, a jumble of nonsense information created by taking the readable information and re-arranging it according to a set of rules.

How effective this process is varies. Let's say that we want to encrypt a short plaintext message, such as

"Hello, how are you?"

We could try shifting all the letters by a certain number of steps in the alphabet; this basic form of encryption is called the *Caesar cipher*.



A basic Caesar cipher with a single right shift.

Shifting all the letters right by 1 position would give us a ciphertext of

"Ifmmp, ipx bsf zpv"

These are definitely not recognisable words, but it also isn't the most complex cipher to break; once the simple rule that encrypts the message is figured out, it is trivial to reverse the operation and recover the message.

This rule, the number of steps that we shift the letters by, can be thought of as the *key* to both performing and reversing the encryption.

We refer to a key which both encrypts and decrypts information as a *symmetric* key. In the cipher above, it's particularly easy to see why. Performing and reversing the encryption follows the same path, like so:

The Caesar cipher is very simple and very easy to break, but there are much more secure symmetric key schemes. One of these, the Advanced Encryption Standard (AES), is frequently used to protect data which is in storage or "at rest". For example, banks use AES to protect customer details stored on their servers so that even if these details are stolen, they're unreadable.

The key that AES uses to encrypt information is much more complex than the one above. It's a numerical value which typically has a length (known as the "security parameter") of 128, 192 or 256 bits, which, at the top end, translates to 2^{256} or

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936

different combinations. This value is used to swap, mix and substitute information in a block in a way which is designed to be easy to reverse if you know the key but impossible if you don't.

There's currently no clever method for finding the key outside of a "brute force" attack where every possible value is checked, a task which is functionally impossible to carry out in a useful time-frame even with access to the most powerful supercomputers. This is considerably more secure than the Caesar cipher, but we *still* can't rely on AES (at least, not on it's own) to securely send a message.

This is because if we want the recipient to be able to read the message when they receive it then we'd need to send the message *and the key!* Anybody intercepting that key could retrieve the message from the encrypted data. We can make this *practically* difficult for an attacker, but not in the same fundamental mathematical way that makes encryption secure.

Not only that, but you'd need to generate a new key if you wanted to communicate securely with someone else, otherwise a legitimate recipient of one key could read *all* of your messages to other people. If you were a big organisation (such as a bank), this would mean managing and protecting millions of individual keys.

We need a different method to keep our messages safe. We need an *asymmetric* key.

Asymmetric keys form the basis of public key cryptography. This is a process that allows anyone to encrypt a message using a "public" key, but only the holder of the "private" key is able to decrypt that message. The two keys do different things, hence the term "asymmetric".

A great (not to mention common) analogy is that of a padlock. If the padlock is initially open, anyone can close the lock to secure a box containing a message, but only the person with the key can open the lock and retrieve the contents.

The thing that was missing from symmetric key cryptography was a secure method for *key distribution*. Public key cryptography gives us the solution.

If two people both create their own public-private key pair and then swap public keys, they can encrypt messages to send to each other (by using the other person's public key) and decrypt messages sent to them by using their own private keys. If the public keys are intercepted, it doesn't matter!

This is also very useful if you need to communicate with millions of people; you only need one public key for any number of people to be able to send you messages, and you only need to store their public keys if you want to send messages back. Even if those keys are stolen, they're not useful. The only thing anybody needs to seriously protect is their single private key.

There are a number of existing schemes for public-key cryptography, but one of the most commonly-used is the Rivest-Shamir-Adleman (RSA) scheme, which has its roots in a very old problem in *number theory.*

Public Key Encryption and Number Theory

Number theory is the branch of mathematics concerned with the properties of whole (integer) numbers. Despite having existed in some form for at least 3800 years, there are many mathematical problems in number theory which still don't have an easy solution. We use the fact that people have been trying to crack these problems for thousands of years as a "proof" that these problems are very hard.

One of these problems is the *prime factoring* problem. The Greek mathematician Euclid proved that any number can be uniquely represented as a product of prime numbers, numbers which only divide by 1 and themselves.

If we multiply two very large (>2048 bit) prime numbers p and q together, we get a larger number N for which p and q are the *unique* prime factors. This is an easy task for a computer, even if the numbers involved are big.

If we are given only N and one of the prime numbers p, it's just as easy to find the other prime q by dividing N by p. This is also easy for computers.

However if we are only given N, finding p and q is hard and takes an impractically long time even for the most powerful supercomputers. Despite the fact that this problem has been known for a very long time, there still isn't an *efficient* process for finding the prime factors of very large numbers!

The prime factoring problem is an example of a *trap-door* function. This is a mathematical operation where the result is very easy to calculate but nearly impossible to reverse *unless* you know at least one non-trivial piece of information.

Of course, having a mathematical operation with these properties is only half the battle. Even if the mathematical basis of an encryption scheme is secure, the actual computer code that executes the scheme may not be. It's not helpful to encrypt data if someone can spy on the workings of your machine and determine the private key as it is generated, so a lot of effort goes into making the practical implementations of cryptography as secure as possible. Huge amounts of information are generated and transmitted by computers, so the computational process of carrying out the encryption and decryption needs to be fast and efficient too.

RSA is based on the prime factoring problem. We won't go *very* deep into the details of RSA here, but when it comes to understanding the threat posed by quantum computing, there are some important ideas that it's helpful to understand.

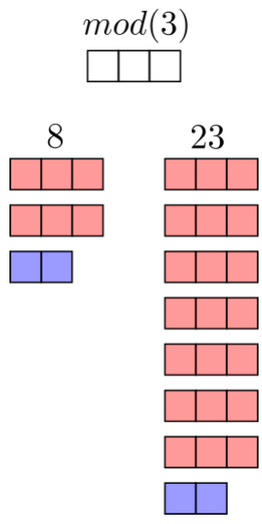
RSA

> *The era of "electronic mail" may soon be upon us; we must ensure that two important properties of the current "paper mail" system are preserved:* (a) *messages are private, and* (b) *messages can be signed.*

So opens the introduction of the original scientific paper of Rivest, Shamir and Adleman, written in 1977, which defined the first *working* example of a public-key cryptosystem (the concept was originally invented by Diffie and Hellman, who created one of the first key-exchange mechanisms). That RSA (with some minor adjustments) is still a dominant public-key cryptosystem today is testament to a winning combination of brilliant cryptographic design and computational efficiency. We'll focus first on the mathematical core of RSA.

Fundamentally, RSA operates in what is known as *modular arithmetic*. If you can interpret a clock, you're familiar with this idea. On a clock, the hours "wrap back" around to the start once you get past either 12 or 24, depending on how your clock displays time. In this case, we call either 12 or 24 the *modulus*.

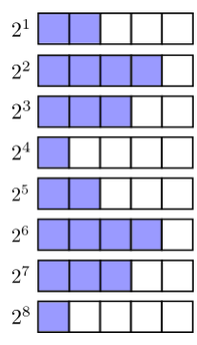
Because we can't go past the modulus (there's no 25 o'clock, for example), when we perform arithmetic operations where the result is larger than the modulus, the result also wraps around. This means that two different numbers can "match up" in a given modulus. We call this matching up *congruence*. For example, 8 and 23 are congruent in modulus 3. We can write this as $8 \equiv 23 \ mod(3)$ where "$\equiv$" indicates congruence.



A visual representation of the numbers 8 and 23 in mod(3). The red boxes show how many times 3 fits into each number, while the blue boxes indicate the remainder. In mod(3), 8 and 23 have the same remainder and are thus congruent.

In RSA, we pick two large primes and multiply them together to create a large number N that we use as a modulus. Information is encrypted by converting each character in a message into a number m (which is how computers internally represent characters anyway; they're translated into human-readable text through standards such as ASCII), and then raising that number to the power of a number e. This produces a ciphertext $c \equiv m^e \ mod \ (N)$, where c is the result of a large number m^e wrapping around the modulus N one or more times and stopping on a different value.

Something interesting happens in modular arithmetic when we raise a given number to an exponent. As the exponent increases, the remainder in the modulus starts to take on a repeating pattern. Because this pattern is repeating, we can say that it has a *period*, in much the same way that a sine wave does.



The value of increasing powers of 2 in mod(5) form a repeating pattern with a particular *period*. *If we change the modulus, this pattern and the associated period change too.*

If our message integer m and modulus N are "co-prime" (don't share any common factors beyond 1), then we can guarantee that as the exponent increases indefinitely, the result periodically returns to the original value.

Because exponentiation in modular arithmetic has this periodic behaviour, we can find another number d such that raising m to the power of ed produces a number that wraps around the modulus until it returns to the original value of the message m. We can see this in the image above, where $(2^4)^2 \ mod(5) \equiv 2^8 \ mod(5)$.

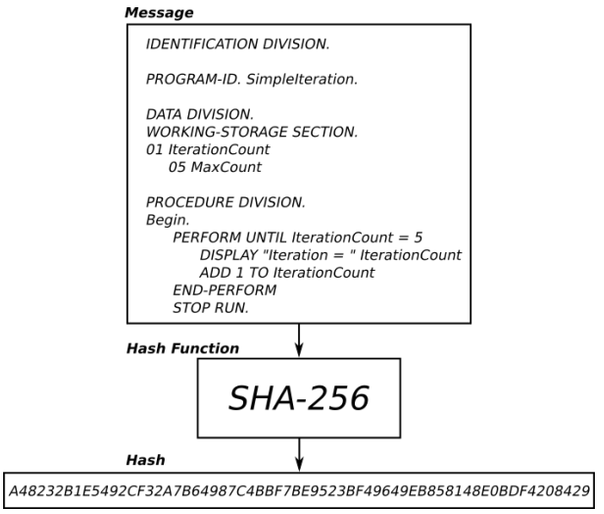
Because $c \equiv m^e mod\ (N)$, raising c to the power of d is the same as writing$(m^e)^d\ mod\ (N) \equiv m\ mod(N)$. Exploiting the properties of congruence and modular exponentiation is what allows us to both encrypt and *decrypt* the message!

In RSA, the public key is the values e and N, while the private key is d. The method for figuring out what e and d should be, how they are connected to N, and the other steps that must be performed to make the scheme truly *secure* are more complicated than we'll get in this article, but you can find a comprehensive description here.

This part of RSA gives us a means of encrypting and decrypting messages, and that alone is very useful. However, this is only half of the full RSA system; the other half is the method used to validate the *origin* of encrypted information, allowing you to be sure that the person sending you a message is who they say they are. This is done through a *digital signature*.

Digital signatures

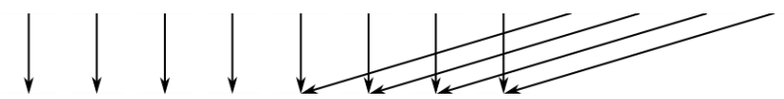
To talk about digital signatures, we'll have to describe another cryptographic tool called a *hash function*. A hash function is a cryptographic process that takes a message of any length as an input, and outputs a value (known simply as a "hash") of fixed length.



An example of a hash function being used to turn a message into a hash. The message can be any data of any length; computer code (as above), a picture, a sound-file...

Hash functions are *deterministic*; given the same input, they should always produce the same output. At the same time, if even a single digital "bit" in the message changes, the hash function should output a completely different value, a property known as the "avalanche effect". Hash functions are very quick to calculate, but it's infeasible to directly reverse the process.

This is partly due to a mathematical argument called the Pigeonhole principle. Ideally we'd like a unique hash for each unique message, but because there are usually far more possible input messages than there are hash outputs of a fixed length, then it is guaranteed that it is possible to generate *collisions* between hash values, in which two messages produce the same hash output. In short, an attack on a hash function isn't done by directly reversing the process, it has to come from another conceptual direction by finding messages that *could* produce that hash.



A (very) simple illustration of the pigeonhole principle. It's clear that, to assign each message (black square) to a pigeonhole (white square), some pigeonholes will have to contain more than one message. Likewise, for a hash function of fixed length, there are only so many hash outputs which can exist, and the number of possible messages is significantly greater than the number of hashes.

Collisions are unavoidable, but a good hash function should still be *resistant*to collisions, in the sense that collisions should be hard to find without using impractically slow and inefficient trial-and-error.

Why does this matter? If we're using hash functions to verify the *content* of a message, then by finding collisions, an attacker could intercept our original message and replace it with a *different* message that still outputs the same hash.

While hash functions have many uses in verifying information (including as proof-of-work in the Bitcoin protocol), one of the most important is in creating digital signatures. This allows the author of a piece of information, be it a message or something more complex (such as a piece of software), to verify that they were the origin and integrity of that information. Here's a basic example of how this works with RSA:

- The sender writes their message and then creates a hash of the message using a specific hash function, such as SHA-256
- The sender then encrypts the hash using their RSA private key, creating a *digital signature*. They add this signature to the message.
- This encryption can (via the public-private key relationship $(m^e)^d\ mod\ (N) \equiv m\ mod(N)$) be reversed using the sender's public key without revealing the private key in the process.
- The sender then creates a single-use key and uses it to encrypt their message using a symmetric key scheme such as DES/Triple-DES or AES
- The sender encrypts this single use key using RSA and the recipient's public key
- They send the RSA-encrypted symmetric key and the DES/AES encrypted message to the recipient.
- The recipient decrypts the symmetric key using their RSA private key, and then decrypts the message and the encrypted hash function using the symmetric key
- The recipient then further decrypts the digital signature using the sender's public key to retrieve the original hash of the message
- The recipient then applies the same hash function to the decrypted message and compares the output against the hash retrieved from the digital signature

- If the hashes match, the recipient can be confident of two things: a) the person who sent the message holds the private key corresponding to the public key used to decrypt the signature, and b) that the message hasn't been tampered with.

In short, the above process is a way for the sender of a message to verify both their identity and message through a specific piece of secret information (their private key) *without* having to directly reveal that information. The same principle also forms the basis of the public-key certificate scheme used to validate the identity of websites on the internet, allowing for secure, trustworthy web browsing.

Why bother with the extra step of using hash functions or symmetric-key encryption when we could also "double-encrypt" the message, once with the sender's private key and then again with the recipient's public key? This would ultimately have the same authentication properties as the above method and is the signature scheme proposed in the original RSA paper.

While RSA works phenomenally well for public key encryption, it does have some limitations. While computers can easily perform the calculations needed for RSA, it's still much more "expensive" than symmetric-key encryption. There's also a maximum length to the message that we can encrypt in one go using RSA, which is not a problem for a symmetric key system that can work on as many "blocks" of information as we like. RSA is therefore used solely as a "key-encapsulation" scheme to securely exchange a single-use *session* key, which is then used with a more efficient symmetric-key "data-encapsulation" scheme to securely send the message.

As you can probably tell by this point, securely sending messages over the internet is *not easy*, and everything we've described so far took many years of development and testing to get right.

Which is why the quantum computing threat is, above all else, rather inconvenient.

The Quantum Threat

Large primes are difficult to factorise using a *classical* computer, and that's what makes schemes based on their properties so secure. The term "classical" means a computer that uses binary logic to perform calculations using bits, individual pieces of information which can be either 1 or 0. This is the type of computer that you're using right now in 2021. If this article is still around in 2040, there's a chance that you might have (or at least have widely-available access to) a machine which is at least in part a *quantum*computer.

While modern digital electronics make use of material properties which are ultimately due to quantum effects (such as semiconductivity), the outcome of an operation using an electronic component is a behaviour from classical physics; the flow of an electrical current.

In a quantum computer, the outcome of an operation using a quantum bit or "qubit" is a quantum *state*, a probability distribution that describes the likely outcome of the measurement of one of the physical properties (e.g the spin) of the system.

This is conceptually quite similar to the operating principles of our optical computing system, in which we encode information into the phase and amplitude properties of light and then allow a physical process to perform the calculation for us. In the case of our optical processor, we leverage the natural Fourier transformation properties of light.

In quantum computing there are several unique properties (such as entanglement) that can be used to perform calculations. The most commonly given example of this is that a qubit can be in a state of quantum superposition with other qubits and be both 0 and 1 *at the same time* until we take a measurement.

Like optical computing, quantum computing has some serious advantages over classical computing when it comes to certain calculations, and a range of quantum algorithms have been proposed that allow us to do things that we couldn't previously.

Unfortunately for cryptography, one of those quantum algorithms is Shor's algorithm.

P vs NP

Not every calculation can be run efficiently on a computer. Those that can are said to run in "polynomial" or "P" time. This means that, for n pieces of information, the time taken to both find a solution to the problem and *verify*that the solution is correct are related to a polynomial of n, such as n^2.

If only the verification can be performed in polynomial time, the problem is said to be *non-deterministic polynomial* or "NP". For example, the time taken to perform a brute-force search for an AES key has an exponential 2^n time dependency where n is the key length. This exponential growth in search time for a linear increase in key length is what makes AES both practical and secure. If you *do* find the key (or have it already), you can still verify that you have been successful by undoing the encryption in polynomial time.

The large-prime factoring problem is an NP problem, in that it takes much longer to find a solution than it does to verify it and is therefore said to be computationally "hard". Shor's algorithm breaks this hardness by converting the factoring problem into a different form in which the answer can be found (in part) by finding the period of an operation carried out in modular arithmetic, the same period that we described above in the RSA cryptosystem!

The conversion part can be done using a classical computer while the unique superposition properties of a quantum computer are needed in order to perform the *quantum* Fourier transform that finds the period.

Using this method, Shor's algorithm is expected to find prime factors in polynomial time. This is a catastrophe for prime-based security; a cryptographic system that can be broken in polynomial time is totally insecure.

And Shor's algorithm isn't the only quantum algorithm that poses a threat to cryptography. Grover's search algorithm provides a means of speeding up a brute-force search such that it only takes root-n operations to search $n$possibilities. This poses a threat to hash functions by reducing the amount of searching that it takes to find a collision.

This even poses a threat to AES, albeit a limited one; Grover's search algorithm can reduce the effective strength of a 256-bit AES key to 128 bits, although at a mere 340,282,366,920,938,463,463,374,607,431,768,211,456 different combinations, that's still comfortably outside of the reach of the world's supercomputers.

There is some good news. Because the kind of quantum computers that can perform Shor's algorithm work directly with quantum states, they are *very*sensitive to noise (such as tiny thermal fluctuations) and it is hard to build a working system with a large number of qubits. A quantum computer with enough qubits to perform Shor's algorithm on a large number doesn't exist yet. Thus far, the largest number to which Shor's algorithm has successfully been applied is 21.

However, these problems are being overcome quickly, and the complexity of quantum computers appears to be growing at a doubly-exponential rate, an observation known as Neven's law. For reference, the phenomenal growth in classical computing power over time reflected in Moore's law is only singly exponential. Quantum computers are not only getting better, they're doing so with almost unreal speed.

And quantum computing is no longer the preserve of highly-specialised research groups. As these systems improve, the infrastructure and learning resources needed to work with them is also improving. Software interfaces and examples for working with quantum computers are now becoming openly accessible. If you want to see the code and mathematical explanation for how Shor's algorithm can be run on a quantum computer, you don't need to dive deep into the scientific literature; you can quite easily find it on Google.

This is great news for quantum computing but bad news for cryptography. The era of quantum supremacy, the tipping point at which quantum computers can find answers to problems which are not possible on any classical computer, will soon be upon us.

In the face of these issues, we need alternative mathematical problems which not only have the same easy-to-perform, hard-to-reverse properties as large prime multiplication but which are also difficult even for quantum computers.

And we need them fast.

Fortunately, help is at hand. The global cryptographic community has risen to the challenge, investigating a wide range of novel quantum-secure techniques and testing them exhaustively for weaknesses. At the time of writing, a range of schemes covering the cryptographic tools needed for message encryption and authentication are undergoing a third and final round of selection by the US National Institute of Standards and Technology (NIST).

Many of these schemes, such as NTRUEncrypt (for public-key cryptography) and NTRUSign (for digital signatures) are derived from lattice-based cryptography. Over the course of our articles on cryptography, we're building towards an explanation as to how we can accelerate these schemes (and more besides) using our Fourier-optical computing system. In our next article, we'll be looking at some of the mathematical problems that make lattice-based cryptography secure.