# McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers

Daniel J. Bernstein, *University of Illinois at Chicago, Ruhr University Bochum;*
Tanja Lange, *Eindhoven University of Technology*

https://www.usenix.org/conference/usenixsecurity20/presentation/bernstein

This paper is included in the Proceedings of the
29th USENIX Security Symposium.
August 12–14, 2020
978-1-939133-17-5

# McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers

Daniel J. Bernstein
*University of Illinois at Chicago, Ruhr University Bochum*

Tanja Lange
*Eindhoven University of Technology*

## Abstract

Recent results have shown that some post-quantum cryptographic systems have encryption and decryption performance comparable to fast elliptic-curve cryptography (ECC) or even better. However, this performance metric is considering only CPU time and ignoring bandwidth and storage. High-confidence post-quantum encryption systems have much larger keys than ECC. For example, the code-based cryptosystem recommended by the PQCRYPTO project uses public keys of 1MB.

Fast key erasure (to provide "forward secrecy") requires new public keys to be constantly transmitted. Either the server needs to constantly generate, store, and transmit large keys, or it needs to receive, store, and use large keys from the clients. This is not necessarily a problem for overall bandwidth, but it is a problem for storage and computation time on tiny network servers. All straightforward approaches allow easy denial-of-service attacks.

This paper describes a protocol, suitable for today's networks and tiny servers, in which clients transmit their code-based one-time public keys to servers. Servers never store full client public keys but work on parts provided by the clients, without having to maintain any per-client state. Intermediate results are stored on the client side in the form of encrypted cookies and are eventually combined by the server to obtain the ciphertext. Requirements on the server side are very small: storage of one long-term private key, which is much smaller than a public key, and a few small symmetric cookie keys, which are updated regularly and erased after use. The protocol is highly parallel, requiring only a few round trips, and involves total bandwidth not much larger than a single public key. The total number of packets sent by each side is 971, each fitting into one IPv6 packet of less than 1280 bytes.

The protocol makes use of the structure of encryption in code-based cryptography and benefits from small ciphertexts in code-based cryptography.

## 1 Introduction

TLS 1.3 highlights the importance of "forward secrecy" by switching completely to Diffie–Hellman-based cryptography for confidentiality. The client initiates the connection and already on the first message sends the preferred cipher suite and a public key. These systems are typically based on elliptic curves though some finite-field options remain. Elliptic-curve keys consume only 32–64 bytes and thus add very little overhead to the packets and computation is very fast, even on small devices.

Unfortunately, if large quantum computers are built then Shor's quantum algorithm [33] breaks ECC in polynomial time. In the two decades since Shor found this quantum speedup, research in cryptography has progressed to find systems that remain secure under attacks with quantum computers. There are several approaches to designing such *post-quantum* systems but the main categories for public-key encryption systems are based on codes, lattices, or—more recently—isogenies between supersingular elliptic curves. Code-based cryptography [23] was invented by McEliece in 1978 and is thus just one year younger than RSA and has held up much stronger against cryptanalysis than RSA. Lattice-based cryptography started more than 15 years later and security estimates are still developing; see, e.g., the recent paper [1] claiming a 400$\times$ speedup in lattice attacks. Isogenies in their current use started only in 2011 [17].

In 2015, the European project PQCRYPTO issued recommendations [2] for confidence-inspiring post-quantum systems; for public-key encryption the only recommended system was a code-based system which is closely related to McEliece's original proposal. However, when in 2016 Google ran an experiment [8] deploying post-quantum cryptography to TLS connections between Chrome (Canary) browsers and Google sites they did not choose a code-based system but a much more recent system based on lattices. The main issue with the high-confidence code-based system is that it requires a much larger key size—1MB vs. 1kB—for the same estimated level of security. Google piggybacked the lattice-based system with ECC so that the security of the combined system would not be weaker than a pure ECC system.

In April 2018, Google's Langley reported [21] on another experiment with post-quantum cryptography, this time testing no particular system but different key sizes. They tested initiation packets of sizes 400, 1100, and 10000 bytes, saying that these are meant to represent systems based on isogenies and based on different types of lattices. Langley justified their choice by writing "in some cases the public-key + ciphertext size was too large to be viable in the context of TLS". There were too many sites that dropped the largest size so it was skipped from further experiments and replaced by 3300 bytes. For these sizes they measured the increase in latency. In a second experiment they measured round-trip times, this time even skipping the 3300-byte size. These sizes are a far cry from what is needed to transmit the 1MB keys of the McEliece cryptosystem. See also the failure reported in [12] to handle 300KB keys in OpenSSL.

For the experiments it is reasonable to use new systems in combination with ECC; see [18–20] for a new lattice-plus-ECC experiment by Google and Cloudflare. However, this does not help with post-quantum security if lattices turn out to be much weaker than currently assumed. This raises the question how network protocols could possibly use the high-confidence McEliece cryptosystem.

Of course, the key could be chopped into pieces and sent in separate packets and the server could be instructed to buffer the pieces and reassemble the pieces but this allows rogue clients to flood the RAM on the server. See Section 2.

This paper introduces McTiny, a new protocol that solves the problem of memory-flooding denial-of-service attacks for code-based cryptography. McTiny handles the 1MB keys of the McEliece cryptosystem, having the same basic data flow as TLS in which the client creates a fresh public key for each connection and sends it as the first step of the protocol. McTiny splits the public keys into pieces small enough to fit into network packets. On the client side the overhead is small compared to creating the key and sending 1MB. The server is not required to allocate any memory per client and only ever needs to process information that fits into one Internet packet, making McTiny suitable for tiny network servers.

Sections 2 and 3 motivate tiny network servers and review existing results. Section 4 gives background in coding theory. Sections 5 and 6 explain our new McTiny protocol. We analyze cryptographic security in Sections 7–9 and present our software implementation and evaluation in Section 10. Finally we consider some alternative choices.

## 2 Server-memory Denial of Service, and the Concept of Tiny Network Servers

Most—but not all!—of today's Internet protocols are vulnerable to low-cost denial-of-service attacks that make a huge number of connections to a server. These attacks fill up all of the memory available on the server for keeping track of connections. The server is forced to stop serving some connections, including connections from legitimate clients. These attacks are usually much less expensive than comparably effective attacks that try to use all available network bandwidth or that try to consume all available CPU time.

### 2.1 A Classic Example: SYN Flooding

The "SYN flooding" denial-of-service attack [14] rose to prominence twenty years ago when it was used to disable an ISP in New York, possibly in retaliation for anti-spam efforts; see [9]. "SYN cookies" [4] address SYN flooding, but from a broader security perspective they are highly unsatisfactory, as we now explain.

Recall that in a normal TCP connection, say an HTTP connection, the client sends a TCP "SYN" packet to the server containing a random 32-bit initial sequence number (ISN); the server sends back a "SYNACK" packet acknowledging the client ISN and containing another random ISN; the client sends an "ACK" packet acknowledging the server ISN. At this point a TCP connection is established, and both sides are free to send data. The client sends an HTTP request (preferably as part of the ACK packet), and the server responds.

The server allocates memory to track SYN-SYNACK pairs, including IP addresses, port numbers, and ISNs. This is exactly the memory targeted by SYN flooding. A SYN-flooding attacker simply sends a stream of SYNs to the server without responding to the resulting SYNACKs. Once the SYN-SYNACK memory fills up, the server is forced to start throwing away some SYN-SYNACK pairs, and is no longer able to handle the corresponding ACKs. The server can try to guess which SYN-SYNACK pairs are more likely to be from legitimate clients, and prioritize keeping those, but a sensible attacker will forge SYNs that look just like legitimate SYNs. If the server has enough SYN-SYNACK memory for $c$ connections, but is receiving $100c$ indistinguishable SYNs per RTT, then a legitimate client's ACK fails with probability at least 99%.

SYN cookies store SYN-SYNACK pairs in the network rather than in server memory. Specifically, the server encodes its SYN-SYNACK pair as an authenticated cookie inside the SYNACK packet back to the client, and then forgets the SYN-SYNACK pair (if it is out of memory or simply does not want to allocate memory). The client sends the cookie back in its ACK packet. The server verifies the authenticator and reconstructs the SYN-SYNACK pair.[1]

[1]For compatibility with unmodified clients, the server actually encodes a very short authenticator inside the choice of server ISN. Modifying both the client and the server would have allowed a cleaner protocol with a longer authenticator. In this paper we are prioritizing security and simplicity above compatibility, so we do not compromise on issues such as authenticator length.

## 2.2 Why Stopping SYN Flooding is Not Enough

SYN cookies eliminate the server's SYN-SYNACK memory as a denial-of-service target: a forged SYN simply produces an outgoing SYNACK[2] and does not interfere with legitimate clients. But what happens if the attacker continues making a connection, not merely sending a SYN but also responding to the resulting SYNACK and sending the beginning of an HTTP GET request? The server allocates memory for the established TCP connection and for the HTTP state, much more memory than would have been used for a SYN-SYNACK pair. The attacker leaves this connection idle and repeats, consuming more and more server memory. Again the server is forced to start throwing away connections.

There is some entropy in the SYNACK that needs to be repeated in the ACK. An attacker who sends blind ACKs will only rarely succeed in making a connection, and these occasional connections will time out before they fill up memory. However, an *on-path attacker*, an attacker who controls any of the machines that see the SYNACKs on the wire or in the air, has no trouble forging ACKs. Forcing attackers to be on-path might deter casual attackers but will not stop serious attackers (see, e.g., [15]).

## 2.3 Tiny Network Servers

As mentioned above, not all Internet protocols are vulnerable to these memory-filling denial-of-service attacks. Consider, for example, a traditional DNS server running over UDP. This server receives a UDP packet containing a DNS query, immediately sends a UDP packet with the response, and forgets the query. A careful implementation can handle any number of clients without ever allocating memory.

DNS has an optional fallback to TCP for responses that do not fit into a UDP packet. However, at many sites, all DNS responses are short. Clients requesting information from those sites do not need the TCP fallback;[3] an attacker denying TCP service will not deny DNS service from those sites. The bottom line is that DNS can, and at some sites does, serve any number of clients using a constant amount of server memory.

Another classic example is NFS, Sun's Network File System [28]. NFS (without locks and other "stateful" features) was explicitly designed to allow "very simple servers" for robustness [28, Section 1.3]:

> The NFS protocol was intended to be as stateless as possible. That is, a server should not need to maintain any protocol state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a failure. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed, or the network temporarily went down. The client of a stateful server, on the other hand, needs to either detect a server failure and rebuild the server's state when it comes back up, or cause client operations to fail.
>
> This may not sound like an important issue, but it affects the protocol in some unexpected ways. We feel that it may be worth a bit of extra complexity in the protocol to be able to write very simple servers that do not require fancy crash recovery.

An NFS server receives, e.g., a request to read the 7th block of a file, returns the contents of the block, and forgets the request. An important side effect of this type of server design is that malicious clients cannot fill up server memory.

This paper focuses on **tiny network servers** that handle and immediately forget each incoming packet, without allocating any memory. The most obvious application is making information publicly available, as in traditional DNS, anonymous read-only NFS, and anonymous read-only HTTP; as DNS and NFS illustrate, it is possible to design protocols that handle this application with tiny network servers. The concept of a tiny network server also allows more complicated computations than simply retrieving blocks of data. Tiny network servers are not necessarily connectionless, but the requirement of forgetting everything from one packet to the next means that the server has to store all connection metadata as cookies in the client. Tiny servers are not necessarily stateless, and in fact the protocol introduced in this paper periodically updates a small amount of state to support key erasure ("forward secrecy"), but we emphasize that this is not per-client state.

Tiny network servers are compatible with reliable delivery of data despite dropped packets: for example, DNS clients retry requests as often as necessary. Tiny network servers are also compatible with congestion control, again managed entirely by the client. Tiny network servers provide extra robustness against server power outages; trivial migration of connections across high-availability clusters of identically configured servers; and the ability to run on low-cost "Internet of Things" platforms.

## 3 The Tension Between Tiny Network Servers and Further Security Requirements

The obvious security advantage of designing a protocol to allow tiny network servers—see Section 2—is that these servers are immune to server-memory denial of service.

[2]Amplifying a packet into a larger packet raises other denial-of-service concerns, but the outgoing SYNACK is not much larger than the SYN.

[3]DNS-over-TCP was also in heavy use for an obsolete ad-hoc high-latency low-security replication mechanism (periodic client-initiated "DNS zone transfers"), but anecdotal evidence suggests that most sites have upgraded to more modern push-style server-replication mechanisms, for example using rsync over ssh.

What is not clear, however, is that tiny network servers are compatible with other security requirements. The pursuit of other security requirements has created, for example, DNS over TLS and DNS over HTTPS, and all implementations of these protocols allow attackers to trivially deny service by filling up server memory, while the original DNS over UDP allows tiny network servers that are not vulnerable to this attack.

In this section we highlight three fundamental security requirements, and analyze the difficulty of building a tiny network server that meets these requirements. We explain how to combine and extend known techniques to handle the first two requirements. The main challenge addressed in the rest of this paper is to also handle the third requirement, post-quantum security.

## 3.1 Requirements

Here are the three requirements mentioned above:

- We require all information to be encrypted and authenticated from end to end, protecting against interception and forgery by on-path attackers.
- We require keys to be erased promptly, providing some "forward secrecy". For comparison, if key erasure is slow, then future theft of the server (or client) allows an attacker to trivially decrypt previously recorded ciphertext.
- We require cryptography to be protected against quantum computers.

Typical cryptographic protocols such as HTTPS handle the first two requirements, and are beginning to tackle the third. However, these protocols create several insurmountable obstacles to tiny network servers. For each active client, the server has to maintain per-client state for a TCP connection, plus per-client state for a TLS handshake followed by TLS packet processing, plus per-client state for HTTP.

We therefore scrap the idea of staying compatible with HTTPS. We instead focus on the fundamental question of whether—and, if so, how—a tiny network server can provide all of these security features.

## 3.2 Cookies Revisited

One approach is as follows. Aura and Nikander [3] claim to straightforwardly "transform any stateful client/server protocol or communication protocol with initiator and responder into a stateless equivalent", and give some examples. The "Trickles" network stack from Shieh, Myers, and Sirer [31,32] stores all of the server's TCP-like metadata as a cookie, and also provides an interface allowing higher-level applications to store their own state as part of the cookie. Why not apply the same generic transformation to the entire per-connection HTTPS server state $X$, straightforwardly obtaining a higher-availability protocol where a tiny network server stores $X$ as a cookie on the client?

The problem with this approach, in a nutshell, is packet size. These papers assume that a client request and a cookie fit into a network packet. Consider, for example, the following comment from Shieh, Myers, and Sirer: "Of course, if the server needs lengthy input from the client yet cannot encode it compactly into an input continuation, the server application will not be able to remain stateless."

Concretely, the Internet today does not reliably deliver 1500-byte packets through IPv4, and does not reliably deliver 1400-byte packets through IPv6 (even when IPv6 is supported from end to end). Normally the lower layer actually delivers 1500-byte packets, but tunnels sometimes reduce the limit by a few bytes for IPv4, and usually reduce the limit by more bytes for IPv6; see, e.g., [29] and [22].

These limits are actually on fragment size rather than end-to-end packet size. Why not split larger packets into fragments? The answer is that this is unacceptable for a tiny network server. Fragments often take different amounts of time to be delivered, so the server is forced to allocate memory for fragments that have not yet been reassembled into packets. This memory is a target of denial-of-service attacks. The only safe solution is to limit the packet size to the fragment size.

IPv6 guarantees that 1280-byte packets (and smaller packets) can be sent from end to end, without fragmentation. This guarantee simplifies protocol design. Historically, some network links had even smaller packet-size limits, and technically the IPv4 standard still allows routers to split packets into much smaller fragments, but it is difficult to find evidence of problems with 1280-byte packets on the Internet today. This paper focuses on clients and servers connected by a network that delivers 1280-byte packets.

It is not entirely inconceivable that all essential details of an HTTPS state could be squeezed into such a small packet, with enough restrictions and modifications to HTTPS. But continuing down this path would clearly be much more work than directly designing a cryptographic protocol for tiny network servers.

## 3.3 ECC For Tiny Network Servers

We instead start from an existing special-purpose cryptographic protocol that *does* work with tiny network servers, namely Bernstein's DNSCurve [5]. This protocol takes advantage of the small size of public keys in elliptic-curve cryptography (ECC), specifically 32 bytes for Curve25519.

A DNSCurve client starts with knowledge of the server's long-term public key $sG$, previously retrieved from a parent DNS server. Here $s$ is an integer, the server's secret key; $G$ is a standard elliptic-curve point; and $sG$ is the output of a mathematical operation, called elliptic-curve scalar multiplication, whose details are not relevant to this paper. The client generates its own public key $cG$, and sends a packet to the server containing $cG$ and the ciphertext for a DNS query. The server immediately responds with the ciphertext for a response, and forgets the query. Both ciphertexts are encrypted and authenticated under a shared secret key, a 256-bit hash of the point $csG$; the server computes $csG$ from $s$ and $cG$, and the client computes $csG$ from $c$ and $sG$. The client knows that the response is from the server: the shared secret key is known only to the client and the server, and nobody else can generate a valid authenticator.

We highlight two limitations of DNSCurve compared to HTTPS, and explain how to fix these. First, each public-key handshake in DNSCurve handles only one query packet and one response packet, while one HTTPS handshake is typically followed by a web page, often 1000 packets or more.

A conceptually straightforward fix is to carry out a separate DNSCurve-style query for each block of a web page. ECC public keys are small, so the traffic overhead is small; ECC operations are very fast, so there is no problem in CPU time. However, our goal is actually to upgrade to post-quantum cryptography, which uses much larger keys, creating performance problems; see below.

A more efficient fix is for the server to encrypt and authenticate the 256-bit shared secret key under a key known only to the server, obtaining a cookie. The server includes this cookie in the response to the client. The server then accepts this cookie as an alternative to $cG$, allowing the client to carry out subsequent queries without sending $cG$ again.

The second limitation that we highlight is the lack of forward secrecy in DNSCurve. DNSCurve clients can erase keys promptly, for example discarding $cG$ after one connection, but it is not so easy for a DNSCurve server to move from one long-term key to another: this requires uploading the new key to the parent DNS server, something that could be automated in theory but that is rarely automated in practice.

One fix is for the client to encrypt its confidential query only to a short-term server key, rather than to the server's long-term key. This takes two steps: first the client issues a non-confidential query asking for the server's short-term public key; then the client issues its confidential query to the server's short-term public key. The server frequently replaces its short-term public key with a new short-term public key, erasing the old key.

What makes these protocols easy to design, within the constraint of tiny network servers, is the fact that ECC keys and ciphertexts fit into a small corner of a network packet. The question addressed in the rest of this paper is whether tiny network servers can achieve encryption, authentication, and key erasure for much larger post-quantum keys.

## 4 Code-Based Cryptography

This section explains the basics of code-based cryptography and specifies the parameters used in this proposal.

McEliece introduced code-based cryptography in 1978 in [23]. The system uses error correcting codes and the public and private keys are different representations of the same code; the private one allows efficient decoding while the public one resembles a random code which makes it hard to decode. In 1986, Niederreiter [25] published a modification of the McEliece scheme which decreases the ciphertext size. Niederreiter's original proposal involved some codes which turned out to be weak, but using Niederreiter's short ciphertexts with the binary Goppa codes [16] proposed by McEliece in his encryption scheme combines the benefits of both schemes.

McBits, by Bernstein, Chou, and Schwabe [7], extends this public-key primitive into a full IND-CCA2 secure encryption scheme, combining it with a KEM-DEM [13] construction. The PQCRYPTO recommendations [2] include McBits as the only public-key encryption scheme. McBits uses a code of length $n = 6960$, dimension $k = 5413$ and adding $t = 119$ errors. These parameters (explained below) lead to a public key of roughly 1MB and to a ciphertext length just $n-k = 1547$ bits. The same parameters are included in the Classic McEliece submission [6] to NIST's call for Post-Quantum systems [26] as `mceliece6960119`. Classic McEliece has been selected by NIST as a Round-2 candidate [27]. Similar considerations as given in the next sections hold for other parameters and other code-based systems with large public keys.

This section explains how key encapsulation and decapsulation work; for details on key generation see [6]. The description is independent of the exact parameters; we use these for illustration purposes whenever concrete sizes are necessary and because these parameters are recommended for long-term security.

The codes considered in this paper are binary codes, meaning that all entries are in $\{0,1\}$ and that computations follow the rules of $\mathbb{F}_2$, i.e., $0+0 = 1+1 = 0, 0+1 = 1, 1\cdot 1 = 1$, and, as always, $0\cdot a = 0$ for any $a$.

### 4.1 Public and Private Keys

The public key is a binary matrix $K = (I|K')$ with $n$ columns and $n-k$ rows. The leftmost $(n-k)\times(n-k)$ part is the identity matrix $I$. The $(n-k)\times k$ matrix $K'$ differs per user.

The private key is an efficient decoding mechanism for the code related to $K$. The decoding details do not matter for this paper but the private key is much smaller than the public key. Key generation is more computationally intensive than encapsulation or decapsulation.

**Example 1** We now introduce a small example with $n = 7$ and $k = 4$ which we will use for the following sections. Let

$$K = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix},$$

then

$$K' = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

## 4.2 Encapsulation and Decapsulation

The basic operation in encapsulation is to compute $K \cdot e$, where $e$ is a randomly chosen binary vector of length $n$ which has weight $t$, i.e., exactly $t$ nonzero entries; and $\cdot$ denotes normal matrix-times-vector multiplication over $\mathbb{F}_2$. The result of this computation is a binary vector $c = Ke$ of length $n-k$. This computation takes the first row $k_1$ of $K$ and computes the dot product with $e$, resulting in the first bit of $Ke$, takes the second row $k_2$ to similarly produce the second bit of $Ke$, etc.

**Example 2** Continuing in the setting of Example 1 and choosing $e = (0,1,0,0,0,1,0)^{\perp}$ gives $c = (0,0,1)^{\perp}$, the sum of the second and the sixth column.

Decapsulation uses the private representation of the code to recover the vector $e$ from $Ke$. As can be seen in the example, $e$ is not unique. The same $c$ is obtained by adding the fourth and the seventh column, or just by taking the third column. The cryptosystem restricts the weight $t$ of $e$ so that $e$ is unique.

## 4.3 Security of Code-Based Cryptography

The cryptographic hardness assumption is that it is hard to find $e$ given $K$ and $c$ for $e$ of fixed (small) weight $t$. This is the syndrome decoding problem which is a hard problem in coding theory for random codes. For a small example like Example 1 it is easy to check all possibilities of low-weight vectors $e$ but the complexity of these attacks grows exponentially with $n$ and $t$. For code-based cryptography based on binary Goppa codes the key-size $(n-k)\cdot k$ grows with the security level $\lambda$ (meaning an attacker takes $2^{\lambda}$ operations) as $(c_0 + o(1))\lambda^2(\lg\lambda)^2$, with $c_0 \approx 0.7418860694$, for the best attacks known today. See e.g. the documentation of [6] for an overview of attacks.

## 4.4 IND-CCA2 Security

The Classic McEliece system includes key confirmation and computes $\mathsf{ENC}(K) = (c, C, S)$ with $c = K \cdot e$, $C = \mathsf{hash}(2,e)$, and $S = \mathsf{hash}(1,e,c,C)$. The pair $(c,C)$ is the ciphertext and $S$ is the shared symmetric key. Here $\mathsf{hash}$ is a cryptographic hash function.

Decapsulation in Classic McEliece computes $\mathsf{DEC}(c,C,\mathsf{sk}) = S$, where $S = \mathsf{hash}(1,e,c,C)$ if the recovered $e$ has the correct weight and satisfies $C = \mathsf{hash}(2,e)$. Else $S = \mathsf{hash}(0,v,c,C)$, where $v$ is a secret value stored for this purpose. Thus, decapsulation never fails. Subsequent steps use $S$ in authenticated encryption, so invalid ciphertexts will produce failed authenticators.

## 5 McTiny Public Keys

This section explains the mathematical details of how the McTiny protocol (described in the next section) can work on pieces of the public key while obtaining correct encryptions.

## 5.1 Partitioning of Public Keys

McTiny transmits a public key $K$ from the client to the server. It splits $K$ and uses that the computation $Ke$ can naturally be composed using parts of $K$ and $e$. Let $K = (I|K')$ and write

$$K' = \begin{pmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,\ell} \\ K_{2,1} & K_{2,2} & K_{2,3} & \dots & K_{2,\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{r,1} & K_{r,2} & K_{r,3} & \dots & K_{r,\ell} \end{pmatrix},$$

where the submatrices $K_{i,j}$ are chosen to be approximately equally sized and small enough to fit into a network packet along with other message parts described in the next section. For ease of exposition assume that each $K_{i,j}$ has $x$ columns and $y$ rows, so $k = x \cdot \ell$ and $n-k = y \cdot r$; in general the pieces may have different sizes as specified by the system parameters. All users use the same values for $n$, $k$, $t$, $\ell$ and $r$, so the size of each $K_{i,j}$ is universally known.

The client transmits $K$ by sending $K_{i,j}$ and the position $(i,j)$ for $1 \le i \le r$, $1 \le j \le \ell$. Upon receipt of $K_{i,j}$ the server computes the partial result $c_{i,j} = K_{i,j}e_j$, where $e_j$ denotes the matching part of $e$ and $c_{i,j}$ the matching part of the resulting vector $c$. For example, $c_{1,\ell} = K_{1,\ell}e_{\ell}$ takes $e_{\ell}$ as the last $x$ positions of $e$, and computes the matrix-vector multiplication $K_{1,\ell}e_{\ell}$ resulting in the length-$y$ vector $c_{1,\ell}$. The first $y$ coordinates of $c$ are given by $c_1 = e_{1,0} + c_{1,1} + c_{1,2} + \cdots + c_{1,\ell}$, with $e_{1,0}$ the first $y$ positions of $e$.

**Example 3** In the setting of Example 1 submatrices may be chosen as $K_{1,1} = (1\ 1), K_{1,2} = (0\ 1), K_{2,1} = (1\ 0), K_{2,2} = (1\ 1), K_{3,1} = (0\ 1)$, and $K_{3,2} = (1\ 1)$. The vector $e = (0,1,0,0,0,1,0)^{\perp}$ gets split into $e_{1,0} = (0), e_{2,0} = (1), e_{3,0} = (0), e_1 = (0,0)^{\perp}, e_2 = (1,0)^{\perp}$. Then $c_1 = e_{1,0} + c_{1,1} + c_{1,2} = (0) + (1\ 1)(0,0)^{\perp} + (0\ 1)(1,0)^{\perp} = 0$, matching the first coordinate of $c$ computed earlier.

Note that each part $c_{ij}$ poses a decoding problem for $e_j$ which is much easier than breaking McEliece. It is thus important that these pieces are cryptographically protected.

## 5.2 Optimization

The partial computations of $c_{i,j}$ are independent of one another and can be performed in any order. These intermediate results take only $y$ bits and are thus much smaller than the $xy$-bit sized parts of $K'$ that they cover.

We define a concrete example `mctiny6960119` of McTiny, using the `mceliece6960119` parameters mentioned above with $k = 5413$ and $n - k = 1547$. To minimize the size of intermediate results we could take $y = 1$, $x = 5413$, and $\ell = 1$, i.e., we could transmit one row of $K'$ at once. However, this would require 1547 steps alone in the stage of sending $K_{i,j}$. Using $\ell = 2$ and combining three rows produces chunks that might be too large for the network packets. Observing that 1MB requires about a thousand packets of 1KB each and aiming for a regular pattern of combination, we opt for $\ell = 8$ and $r = 119$ for `mctiny6960119`. Typical $K_{i,j}$ then have 13 rows and 680 columns fitting into 1105 bytes. Replies with $c_{ij}$ fit into 2 bytes.

## 6 The McTiny Protocol

This section introduces the McTiny protocol. Forward secrecy is achieved by requiring each client to generate and send a fresh public key $K$ to the server. Clients are also responsible for key erasure at the end of a session. McTiny makes it possible for the server to compute $Ke$, for a big matrix $K$ and chosen weight-$t$ vector $e$ (see Section 4), without requiring the server to allocate any per-client memory and without needing more temporary memory than what fits into a network packet. At the end of the McTiny protocol, server and client both compute their shared symmetric key. The details of how this shared key is computed match Classic McEliece [6] and the client can use decapsulation from Classic McEliece.

Besides code-based cryptography for the public-key operations, McTiny uses authenticated encryption with symmetric keys. The PQCRYPTO recommendations [2] suggest either AES-GCM with AES-256 or Salsa20-Poly1305 with 256-bit encryption keys and 128-bit authentication keys. McTiny follows McBits in using XSalsa20-Poly1305. (XSalsa20 handles longer nonces than Salsa20.) We use $\mathrm{AE}(T : N : S)$ to denote the authenticated encryption of $T$ under key $S$ using nonce $N$.

### 6.1 General Setup and Phases

The server has a long-term public key $\mathsf{pk}$ which is used to authenticate the server and to derive a shared secret key to encrypt and authenticate all messages after the initiation message. The McTiny protocol uses a long-term McEliece key to achieve full post-quantum security. The server administrator generates this long-term key when setting up the server. The public-key infrastructure, the mechanism that disseminates and certifies server public keys, is outside the scope of this paper; we simply assume that, when the McTiny protocol begins, the client knows the server's long-term public key.

The McTiny protocol runs in four phases. The first phase, phase 0, is the initiation phase in which the client establishes contact with the server and the server proves its identity. Specifically, the server uses its long-term private key $\mathsf{sk}$ to decrypt the key $S$ encapsulated by the client and respond to the client's initial request. Note that this key $S$ is not forward-secret. The client and the server use $S$ to encrypt and authenticate all following messages.

In phase 1 the client sends the matrix parts $K_{i,j}$ to the server and the server replies with encryptions of the partial encryptions $c_{i,j}$. Phase 2 is the row-wise combination and phase 3 computes the KEM ciphertext. A full description of the protocol is given in Figure 1. The following sections explain the steps in detail. See Table 1 for the packet sizes in each phase.

### 6.2 Nonces

XSalsa20-Poly1305 uses nonces with 24 bytes. In the McTiny protocol the server is responsible for generating a random 22-byte $N$ from which most nonces are generated as $n = (N, N_0, N_1)$ in a deterministic way. Bytes $N_0$ and $N_1$ are determined by the phase the protocol is in, information regarding positions, and $N_0$ is even for messages from the client to the server and odd for messages the other way. Bytes are stated as integers in $[0, 255]$ in the protocol. For concreteness we state particular choices of $(N_0, N_1)$ below for the `mctiny6960119` parameters. These apply to a large range of codes.

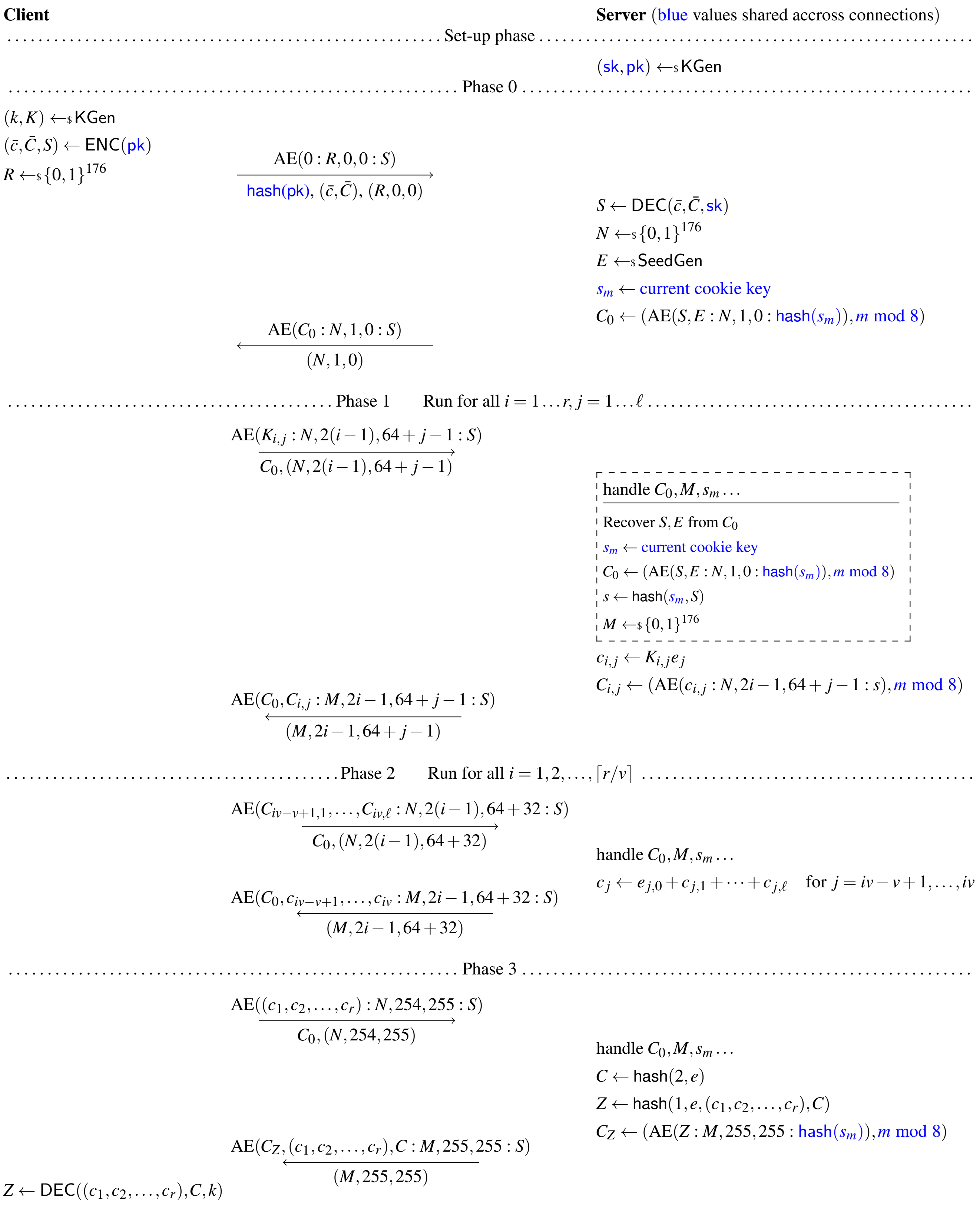
### 6.3 Server Cookies

McTiny makes heavy use of encrypted cookies to store intermediate results in the network/on the client's computer. The cookie keys $s_m$ are symmetric keys that are used for certain time intervals. In time interval $m$ the server uses $s_m$ to encrypt cookies with data to itself. The server can decrypt cookies returned to it during $z$ time intervals, while it is using $s_m, s_{m+1}, s_{m+2}, s_{m+3}, \ldots, s_{m+z-1}$. When the server generates $s_{m+z}$ it erases $s_m$.

In `mctiny6960119` we specify the time interval as one minute and specify that the server remembers 8 cookie keys in any time interval, i.e. while it uses $s_m$ it also remembers $s_{m-1}, s_{m-2}, \ldots s_{m-7}$ but not $s_{m-8}$ or earlier keys. Each cookie contains one byte in clear which determines the cookie index modulo 8, where numbers are assigned round robin. At 22 bytes, the nonce part $N$ is chosen long enough so that it will not repeat while the same cookie key is in use. To explain these choices, assume keys are erased within 8 minutes. If the server uses only 2 keys then client connections begin failing after a 4-minute network outage. Increasing 2 to 8 cheaply increases 4 minutes to 7 minutes. We allocate an entire byte (and add a random multiple of 8 as grease, and have the client repeat the entire byte) so that modifications to the cookie policy do not require modifications to clients.

### 6.4 Phase 0: Initiation

To initiate communication, the client uses the server's long-term public key $\mathsf{pk}$ and derives a symmetric key $S$ using the



**Client** **Server** (blue values shared accross connections)

Set-up phase

$(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}$

Phase 0

$(k, K) \leftarrow_{\$} \mathsf{KGen}$

$(\bar{c}, \bar{C}, S) \leftarrow \mathsf{ENC}(\mathsf{pk})$

$R \leftarrow_{\$} \{0,1\}^{176}$

$$\xrightarrow[\mathsf{hash(pk)},\ (\bar{c},\bar{C}),\ (R,0,0)]{\text{AE}(0 : R, 0, 0 : S)}$$

$S \leftarrow \mathsf{DEC}(\bar{c}, \bar{C}, \mathsf{sk})$

$N \leftarrow_{\$} \{0,1\}^{176}$

$E \leftarrow_{\$} \mathsf{SeedGen}$

$s_m \leftarrow$ current cookie key

$C_0 \leftarrow (\text{AE}(S, E : N, 1, 0 : \mathsf{hash}(s_m)), m \bmod 8)$

$$\xleftarrow[(N,1,0)]{\text{AE}(C_0 : N, 1, 0 : S)}$$

Phase 1 Run for all $i = 1 \ldots r, j = 1 \ldots \ell$

$$\xrightarrow[C_0,\ (N, 2(i-1), 64+j-1)]{\text{AE}(K_{i,j} : N, 2(i-1), 64+j-1 : S)}$$

handle $C_0, M, s_m \ldots$

Recover $S, E$ from $C_0$

$s_m \leftarrow$ current cookie key

$C_0 \leftarrow (\text{AE}(S, E : N, 1, 0 : \mathsf{hash}(s_m)), m \bmod 8)$

$s \leftarrow \mathsf{hash}(s_m, S)$

$M \leftarrow_{\$} \{0,1\}^{176}$

$c_{i,j} \leftarrow K_{i,j} e_j$

$C_{i,j} \leftarrow (\text{AE}(c_{i,j} : N, 2i-1, 64+j-1 : s), m \bmod 8)$

$$\xleftarrow[(M, 2i-1, 64+j-1)]{\text{AE}(C_0, C_{i,j} : M, 2i-1, 64+j-1 : S)}$$

Phase 2 Run for all $i = 1, 2, \ldots, \lceil r/v \rceil$

$$\xrightarrow[C_0,\ (N, 2(i-1), 64+32)]{\text{AE}(C_{iv-v+1,1}, \ldots, C_{iv,\ell} : N, 2(i-1), 64+32 : S)}$$

handle $C_0, M, s_m \ldots$

$c_j \leftarrow e_{j,0} + c_{j,1} + \cdots + c_{j,\ell}$ for $j = iv - v + 1, \ldots, iv$

$$\xleftarrow[(M, 2i-1, 64+32)]{\text{AE}(C_0, c_{iv-v+1}, \ldots, c_{iv} : M, 2i-1, 64+32 : S)}$$

Phase 3

$$\xrightarrow[C_0,\ (N, 254, 255)]{\text{AE}((c_1, c_2, \ldots, c_r) : N, 254, 255 : S)}$$

handle $C_0, M, s_m \ldots$

$C \leftarrow \mathsf{hash}(2, e)$

$Z \leftarrow \mathsf{hash}(1, e, (c_1, c_2, \ldots, c_r), C)$

$C_Z \leftarrow (\text{AE}(Z : M, 255, 255 : \mathsf{hash}(s_m)), m \bmod 8)$

$$\xleftarrow[(M, 255, 255)]{\text{AE}(C_Z, (c_1, c_2, \ldots, c_r), C : M, 255, 255 : S)}$$

$Z \leftarrow \mathsf{DEC}((c_1, c_2, \ldots, c_r), C, k)$

Figure 1: The McTiny Protocol. All AE ciphertexts are decrypted and validated. The offsets are chosen for $\ell \leq 32$.

| phase | | bytes/packet | packets | bytes |
|---|---|---|---|---|
| 0 | query | 810 | 1 | 810 |
| | reply | 121 | 1 | 121 |
| 1 | query | 1226 | 952 | 1 167 152 |
| | reply | 140 | 952 | 133 280 |
| 2 | query | 1185 | 17 | 20 145 |
| | reply | 133 | 17 | 2 261 |
| 3 | query | 315 | 1 | 315 |
| | reply | 315 | 1 | 315 |
| | queries | | 971 | 1 188 422 |
| | replies | | 971 | 135 977 |

Table 1: Packet sizes in each phase of `mctiny6960119`, counting only application-layer data and not counting UDP/IP/Ethernet overhead. A public key is 1 047 319 bytes.

Classic McEliece KEM. The client sends the KEM ciphertext, a hash of the server's public key, and $\mathrm{AE}(0:R,0,0:S)$. The plaintext 0 is a 512-byte extension field, currently unused.

The server uses $\mathsf{sk}$ to decapsulate the KEM ciphertext and retrieve $S$. It verifies and decrypts $\mathrm{AE}(0:R,0,0:S)$. It then picks a random seed $E$ of 32 bytes, computes $C_0 = (\mathrm{AE}(S,E:N,1,0:\mathsf{hash}(s_m)),b)$, using the current cookie key $s_m$ and $b = m \bmod 8$. The seed $E$ determines the low-weight vector $e$ through a deterministic function, see [6] for details. The server then picks a 22-byte nonce $N$, and computes and sends $\mathrm{AE}(C_0:N,1,0:S),(N,1,0)$ to the client. At this point the server forgets all data related to this client.

The client verifies the authenticity and stores $(C_0,N)$.

For `mctiny6960119`, $S$ and $E$ have 32 bytes, nonces have 24 bytes, and $\mathrm{AE}(T:N:S)$ has the same length as $T$ plus a 16-byte authentication tag. In total, $C_0$ has $32+32+16+1=81$ bytes and the message to the client has 40 bytes more for the nonce $N,1,0$ and the authenticator under $S$.

### 6.5 Phase 1: Partial Public-Key Encryption

Phase 1 sends the matrix pieces $K_{i,j}$ to the server which then computes the partial matrix-vector products as described in Section 5. Here we detail the cryptographic protections for this computation.

For every partial matrix $K_{i,j}$ the client sends $\mathrm{AE}(K_{i,j}:N,2(i-1),64+j-1:S),C_0,(N,2(i-1),64+j-1)$ to the server, where $1 \le i \le r$ is the row position and $1 \le j \le \ell$ is the column position. The offset 64 works for $\ell \le 32$. This nonce is deterministic within one run and will repeat, but only to resend the same message. The related nonces save bandwidth.

If the server obtained this message before the expiry of cookie key decrypting $C_0$, the server obtains $(S,E)$ and uses $S$ and $N$ to verify and decrypt the payload to obtain $K_{i,j}$. The server recovers $e$ from $E$ and computes the partial matrix-vector multiplication $c_{i,j} = K_{i,j}e_j$. The position of this matrix is computed from the nonce. The server then recomputes $C_0 = (\mathrm{AE}(S,E:N,1,0:\mathsf{hash}(s_m)),b)$, using the current cookie key $s_m$ and the same nonce $N,1,0$ as before. Finally, it computes a client-specific cookie key $s = \mathsf{hash}(s_m,S)$ and the cookie matching the partial encryption $C_{i,j} = (\mathrm{AE}(c_{i,j}:N,2i-1,64+j-1:s),b)$. It picks a fresh 22-byte random nonce $M$, sends $\mathrm{AE}(C_0,C_{i,j}:M,2i-1,64+j-1:S),(M,2i-1,64+j-1)$ to the client, and forgets all data related to the client.

The client verifies, decrypts, updates $C_0$, and stores $C_{i,j}$ for future use. If partial encryptions are missing, the client re-requests by sending the same packet with the latest $C_0$.

For `mctiny6960119` the public key is split into $119 \times 8$ blocks so that $119 \cdot 8 = 952$ packets need to be sent. Each packet from client to server has 1226 bytes and each reply from the server has 140 bytes (81 in $C_0$, $2+16+1$ in the $C_{ij}$, 16 in the authenticator, and 24 in the nonce).

### 6.6 Phase 2: Row-wise Combination

This phase combines the partial encryptions row-wise. The protocol specifies a split of the $r \times \ell$ blocks from phase 1 into batches of $w$ blocks. Once the client has obtained all blocks in one batch—this may be part of a row as $C_{i,wJ+1},C_{i,wJ+2},\ldots,C_{i,wJ+w}$ or cover one or several rows as $C_{iv-v+1,1},C_{iv-v+1,2},\ldots,C_{iv-v+1,\ell},\ldots,C_{iv,1},C_{iv,2},\ldots,C_{iv,\ell}$ for $v = w/\ell$—it sends them for partial combination. For simplicity and because it matches the sizes of `mctiny6960119` we describe the case where several complete rows of blocks can be handled in one step. For rows $iv-v+1$ through $iv$ the client sends $\mathrm{AE}(C_{iv-v+1,1},C_{iv-v+1,2},\ldots,C_{iv,\ell}:N,2(i-1),64+32:S),C_0,(N,2(i-1),64+32)$. The nonce is separated from the other nonces as $\ell \le 32$.

The server checks the authenticator and, for each $j$ from $iv-v+1$ through $iv$, decrypts $C_{j,1},\ldots,C_{j,\ell}$ to obtain the pieces $c_{j,1},\ldots,c_{j,\ell}$ of $c_j$. As described in Section 5, the server computes $c_j = e_{j,0}+c_{j,1}+c_{j,2}+\cdots+c_{j,\ell}$, with $e_{j,0}$ the matching $y$ positions of $e$. Finally it sends $\mathrm{AE}(C_0,c_{iv-v+1},c_{iv-v+2},\ldots,c_{iv}:M,2i-1,64+32:S),(M,2i-1,64+32)$.

The client verifies, decrypts, updates $C_0$, and stores $c_{iv-v+1},c_{iv-v+2},\ldots,c_{iv}$ for future use. As before, missing pieces are re-requested.

In `mctiny6960119` $v=7$ rows of blocks (i.e., 91 rows from the original matrix) are handled together. Thus, nonces from the client to the server have $N_0 \in \{0,2,4,\ldots,32\}$. Messages from client to server have 1185 bytes, messages from server to client have 133 bytes.

### 6.7 Phase 3: Decapsulation

Eventually all $c_i, 1 \le i \le r$, are known to the client. To match the Classic McEliece key derivation, McTiny has the client send $c$ to the server. The server computes the plaintext confirmation $C = \mathsf{hash}(2,e)$, shared secret $Z = \mathsf{hash}(1,e,c,C)$ and shared-key cookie $C_Z = \mathrm{AE}(Z:M,255,255:\mathsf{hash}(s_m))$ for a fresh nonce $M$. The server sends $(\mathrm{AE}(C_Z, c, C : M, 255, 255 : S), (M, 255, 255))$ to the client which then computes $Z = \mathsf{DEC}(c, C, k)$ and stores $Z$ and cookie $(C_Z, M)$ for future use. The client erases all other data including $S$ and $(k, K)$.

In `mctiny6960119` client and server each send 315 bytes in this phase. In total, 971 packets are sent by the client to the server, each prompting a reply fitting into one packet and never amplifying the size.

The McTiny key exchange ends at this point, and the client communicates securely with the server using session key $Z$. Details of a session protocol are outside the scope of this paper, but the main point is that the client can include $(C_Z, M)$ in subsequent packets so that the server can reconstruct $Z$. Any packet back includes an updated cookie $(C_Z, M)$ using a fresh nonce; the session protocol can update $Z$ here for prompt key erasure within a session. When the session ends (explicitly or by a timeout), the client erases all data.

## 7 Key Erasure

Key erasure should prevent an attacker learning secrets from past connections if he steals the server or client or both.

### 7.1 Key Erasure On the Server Side

An attacker who steals a server obtains the server's long-term secret key $\mathsf{sk}$. If the attacker keeps the server online it can decrypt all future traffic to the server, and can pose as the server. The question is whether the attacker can also decrypt past traffic. Traffic is encrypted under the key $Z$ exchanged by the McTiny protocol, so the question is whether the attacker learns $Z$ for past connections.

The secret key $\mathsf{sk}$ allows the attacker to decapsulate all KEM messages ever sent to the server (phase 0) and obtain all shared keys $S$. These decrypt all messages sent between client and server in subsequent phases under the respective $S$. In particular, the attacker sees a McEliece ciphertext $(c, C)$ sent by the server to the client. However, unless there is a security problem with the McEliece system, the attacker cannot determine $Z$ from this ciphertext.

By stealing the server, the attacker also learns the server's *recent* cookie keys $s_m, \dots, s_{m-z+1}$. In `mctiny6960119` the attacker obtains the cookie keys used for the last 8 minutes. The cookie keys allow the attacker to decrypt cookies under the last $z$ keys from the server to itself; in particular, the attacker can obtain $Z$ for any *recent* connection by decrypting the cookie $C_Z$. However, the attacker cannot decrypt older cookies. Cookies are often repeated across packets, but this linking of packets is already clear from simple traffic analysis.

Here is what the attacker sees for an older McTiny connection, a connection that completed more than $z$ intervals before the theft: the client's short-term McEliece public key $K$ (in blocks $K_{i,j}$); a random ciphertext $(c, C)$ sent to this key, communicating a secret key $Z$ to the client; and the cookies $C_0$ and $C_{i,j}$ for $1 \le j \le \ell, 1 \le i \le r$.

The shared secret $Z$ could be computed from $E$ included in $C_0$, but the keys to all the $C_0$ cookies are erased.

Each $c_{ij}$ includes information on $e$ as a much simpler decoding problem, but the $c_{ij}$ are encrypted in the $C_{ij}$ under erased cookie keys.

### 7.2 Keep Alive

An attacker planning to steal a server in the future has an interest in keeping a connection alive by replaying messages from the client. The client messages include $C_0$ or $C_Z$ in plain and a replay will prompt the server to reply as long as these outer cookies can be decrypted. Each reply includes a fresh $C_0$ or $C_Z$ but these cookies are superencrypted under $S$ or $Z$ which the attacker does not know, yet.

The client is assumed to maintain state, so will no longer reply (and provide fresh versions of $C_0$ or $C_Z$) after the connection was closed. The attacker loses the ability to cause replies after the last cookie expired. Thus an active attacker can extend the lifetime by $z - 1$ time intervals.

In `mctiny6960119` this means that 15 minutes after the end of a connection even an active attacker cannot recover any short-term keys by stealing the server.

### 7.3 Key Erasure On the Client Side

Similarly, an attacker who steals a client obtains the secret keys that decrypt current connections, but the McTiny client software does not retain the keys for a connection once the connection is over. Of course, other parts of the client system might retain data for longer.

Since the client has state it will not keep a connection open longer than specified by its local timeouts. An active attacker cannot override the client's timeout policy.

## 8 Confidentiality and Integrity

In the absence of server theft, there is an extra layer of protection for confidentiality: all packets from the client are encrypted to the server's long-term McEliece key (phase 0) or use authenticated encryption. The choice of cryptographic primitives follows best practices, so we look for weaknesses introduced by the protocol. We first analyze what an external attacker is faced with and then what a malicious client can do.

### 8.1 Passive External Attacker

Authenticated encryption guarantees confidentiality and integrity, but only if different messages never share nonces. If AE is AES-GCM, and an attacker sees $\mathrm{AE}(T_1 : N : S)$ and $\mathrm{AE}(T_2 : N : S)$ for $T_1 \ne T_2$, then the attacker can also produce authenticators for arbitrary ciphertexts under $(N' : S)$ for any $N'$. Our choice of XSalsa20-Poly1305 for AE limits the impact, but the attacker can still produce authenticators for arbitrary ciphertexts under $(N : S)$. Either way, we need to analyze the potential for nonce reuse.

All packets from the server to the client use authenticated encryption with a fresh nonce under shared key $S$. The random part of the nonce has 22 bytes (176 bits) and thus the choice will not repeat while $S$ is in use. Additionally, the domains for the nonces are separated by step and direction using the last two bytes. If a step is repeated due to packet loss, the server will make a fresh choice of $M$. Hence, the attacker will not observe nonce reuse for different plaintexts.

The subsequent messages from the client to the server are encrypted and authenticated using $S$ and a nonce which depends on the first random nonce $N$ chosen by the server. Again the last two bytes provide domain separation. This makes the choice of nonce deterministic for each encryption and the same nonce and key are used when retransmitting in the same phase, but only to encrypt the same plaintext.

The attacker also sees $C_0 = (\mathrm{AE}(S, E : N, 1, 0 : \mathtt{hash}(s_m)), m \bmod 8)$ using several cookie keys $s_m$ under the same nonce $N, 1, 0$. All cookies encrypt the same message, hence nonce-reuse under the same $s_m$ is no problem. There is no weakness in AE for using the same nonce and plaintext under different keys.

The connection for a different client served by the same server uses a different $S'$ and $N'$. Figure 1 highlights values shared across clients in blue. Messages with the same key either have different nonces or are identical.

### 8.2 Active External Attacker

The Classic McEliece KEM is secure against active attacks, hence the shared secret $S$ is not known to the attacker. Authenticated encryption protects the other packets against active attackers attempting to forge or modify packets. Every ciphertext is verified upon receipt.

Clients and external attackers cannot influence the choice of nonce and any modification of $N$ leads to invalid authenticators and thus to no reply from the server. The client accepts messages under key $S$. Replays of server messages will not cause a reaction from the client as it has state.

Mixing cookies and messages from different clients does not work. The server accepts cookies under its most recent cookie keys $s_m, s_{m-1}, \ldots, s_{m-z+1}$ and uses the symmetric key $S$ provided in $C_0$ to decrypt and check the rest of the message.

The attacker can replay a valid client message and cause the stateless server to handle it again. If the cookie key has changed this leads to a different $C_0'$ in the reply. For the outer encryption a random $M$ (or $N$ in phase 0) is chosen, hence only the last two bytes of the nonce repeat, the rest of the nonce differs, meaning no loss in security.

### 8.3 Malicious client

A malicious client knows $S$ and $Z$ anyway, so its targets are the cookie keys. The following assumes that the client manages to send the attack packets in the same cookie interval. Else more retries are needed.

The encryption of $Z$ uses fresh 22 bytes for the nonce. The computation of $C_0$ is deterministic depending on verified values. Initiating a new connection and thus changing $E$ leads to a fresh choice of $N$.

The malicious client can send $K_{11}$ and $K_{11}'$, likely causing $c_{11} \neq c_{11}'$. This produces $C_{11} \neq C_{11}'$ which use the same nonce and key. The client (as opposed to an external attacker) obtains these cookies. However, the key $s = \mathtt{hash}(s_m, S)$ is used only for this client, limiting the use to forging server cookies for this one step in its own connection. Furthermore, if $K_{11}$ and $K_{11}'$ differ only in the first column, the client learns that the first bit in $e$ is set if $C_{11} \neq C_{11}'$ and else that it is not set. However, the target of the McTiny protocol is for the server to send $e$ to exactly this client.[4]

Note that both of these attempts come at the expense of sending two messages under the same nonce with $S$, giving away the authenticator under that key and nonce. This is not an attack as the client could choose to leak $S$ in its entirety.

## 9 Security Against Quantum Computers

The McTiny protocol is designed to make the well-studied McEliece cryptosystem practical for tiny network servers. All public-key cryptography in the protocol uses this system for its resistance to attacks using quantum computers.

The McTiny protocol is flexible in the parameter choices for the code-based part with minimal adjustments on the number of steps per phase. `mctiny6960119` uses very conservative parameters. This means that even an active attacker with a quantum computer cannot break the public-key encryption.

All of the keys for symmetric cryptography are 32 bytes, providing ample protection against Grover's algorithm and the choice of XSalsa20-Poly1305 for AE follows recommendations for post-quantum security.

## 10 Implementation and Evaluation

This section describes our implementation of the `mctiny6960119` protocol, and evaluates whether the protocol lives up to its promise to run safely on tiny network servers. The implementation is now available at `https://mctiny.org`.

### 10.1 Interface

Our software provides four main tools:

[4]The malicious client learns this bit prematurely, but to learn $e$ it needs about 6960 steps, much more than a regular run of 971 steps would take.

- `master` creates a new `mctiny6960119` server identity: a long-term public key and a long-term secret key.
- `rotate` is run every minute to update the pool of 8 server cookie keys, creating a new cookie key and erasing the oldest cookie key.
- `server` handles the server side of the `mctiny6960119` protocol: it binds to a specified UDP port on a specified local IP address and handles incoming request packets from any number of clients.
- `client` performs one run of the client side of the `mctiny6960119` protocol, communicating to a server at a specified address and port, using a specified server public key.

The decomposition of server-side tools is meant to easily support replicated server deployments as follows. The server administrator runs the `master` tool on a master device, and pushes the results to each server through standard tools such as `rsync`. Each server runs the `server` and `rotate` tools. The master device needs enough resources to generate and store the public key, but each server can be much smaller.

The `master`, `rotate`, and `server` tools manage data in a state directory specified by the caller. The public key is stored in a file `state/public/d53...` where `d53...` is a 256-bit key hash in hex. The secret key is stored in `state/secret/long-term-keys/d53....` The `server` tool transparently supports multiple secret keys for multiple identities on the same host. Cookie keys are stored in `state/secret/temporary-cookie-keys/0` through `state/secret/temporary-cookie-keys/7`, with `state/secret/temporary-cookie-keys/latest` symlinked to the current key.

Our API for each of these tools is the standard UNIX command line, making the tools directly usable from a wide range of languages. The command line involves some overhead to spawn a new process, and obviously the same functions could also be provided through APIs with less overhead, but we have not found evidence that the command line is a performance problem here.

## 10.2 Internals

We reused existing Classic McEliece software [6] for key generation (in `master` for long-term keys, and in `client` for short-term keys), encryption (in `client` for long-term keys), and decryption (in `server` for long-term keys, and in `client` for short-term keys). We also reused existing software for symmetric encryption (XSalsa20), symmetric authentication (Poly1305), and hashing (SHAKE256).

We obtained all of this software from the SUPERCOP cryptographic benchmarking framework. The tests described below use version 20191017 of SUPERCOP, the latest version at the time of this writing.

For our new McTiny software components, we selected the C programming language, with the goal of eliminating unnecessary performance overheads. C is, however, notorious for encouraging devastating bugs, such as memory-safety bugs. We do not have evidence that a Rust rewrite would make the software noticeably slower; it could even be faster. We are also not claiming that achieving any particular performance level is more important than reducing risks of bugs.[5]

We wrote new cryptographic software for `mctiny6960119`'s matrix-partitioning encryption (in `server`), ensuring compatibility of the final result with Classic McEliece. We also wrote new software for the higher-level aspects of `mctiny6960119`, such as packet construction, packet parsing, and the general packet flow. Overall we wrote about 2500 lines of new code; see Table 2. The file `mctiny.h` is output by a 160-line `mctiny.py` that supports variations in McTiny parameters.

## 10.3 RAM Consumption

Running `size` on the compiled `server` binary shows 206586 bytes of code, 792 bytes of initialized data, and 23824 bytes of bss (further RAM initialized to 0 at program startup). See Table 2. The code size includes all of the cryptographic software that we imported from SUPERCOP, such as the Classic McEliece software and the SHAKE256 software. Our code uses only a small amount of stack space, and it avoids all heap allocation and `mmap`-based allocation.

We do not claim that the entire program would work in such a small amount of RAM without modification. The problem is that we also use some OS libraries that were not designed to be small: we use `stdio` for file management and printing error messages, we call `getaddrinfo` to determine the local IPv4 or IPv6 address to use, etc. We found that the server works with stack size limited to 92KB (`ulimit -s 92`) but not with a smaller stack size. Also, monitoring system calls with `strace` shows various memory allocations from standard libraries at program startup.

The `rotate` program uses 920 bytes of initialized data and 944 bytes of bss. The `master` program needs more RAM to create a public key: it uses 752 bytes of initialized data and 1062560 bytes of bss. The `client` program uses 800 bytes of initialized data and 1154648 bytes of bss.

## 10.4 Network Usage

On an unloaded network we saw (as expected—see Table 1) 971 packets from the client to server, plus 971 packets from the server to the client, for a complete `mctiny6960119` key exchange. The packets from client to server occupied a total of 1 188 422 bytes of application-layer data (not counting per-packet bandwidth overhead, which is normally 8 bytes

[5]We did take some steps to reduce these risks, such as running tests under Address Sanitizer.

| text | | data | | bss | | c | h | file | purpose |
|---|---|---|---|---|---|---|---|---|---|
| 155 | 0 | 0 | 0 | 0 | 0 | 13 | 8 | `hash` | SHAKE256 wrapper (copied) |
| 5406 | 0 | 0 | 0 | 24 | 0 | 216 | 72 | `mctiny` | library for McTiny computations |
| 8377 | 209 526 | 0 | 800 | 0 | 1 154 648 | 530 | 0 | `mctiny-client` | connect to a server |
| 2589 | 184 487 | 0 | 752 | 0 | 1 062 560 | 149 | 0 | `mctiny-master` | create a server key |
| 3063 | 22 944 | 104 | 920 | 0 | 944 | 199 | 0 | `mctiny-rotate` | rotate cookie keys once |
| 6538 | 206 586 | 24 | 792 | 32 | 23 824 | 546 | 0 | `mctiny-server` | serve any number of clients |
| 989 | 196 599 | 0 | 656 | 0 | 1 064 952 | 63 | 0 | `mctiny-test` | local keypair/enc/dec test |
| 5313 | 0 | 0 | 0 | 0 | 0 | 612 | 24 | `pacing` | client-side congestion control |
| 1158 | 0 | 8 | 0 | 1284 | 0 | 111 | 20 | `packet` | build and parse packets |

Table 2: Source and object sizes. The "`c`" and "`h`" columns are the number of lines in `file.c` and `file.h`. The "text", "data", and "bss" columns are the sizes reported by the standard `size` tool for the object file `file.o`, and for the linked binary `file` when a second number is reported. Sizes of binaries listed here include sizes for cryptographic software imported from SUPERCOP (e.g., the Classic McEliece software), but do not include sizes for standard shared libraries from the OS (e.g., `getaddrinfo`). Code is compiled for Intel Haswell using `gcc` with optimizations `-O3 -march=native -mtune=native`. Compiler verson is 7.4.0, as shipped with the current long-term-support version of Ubuntu (Ubuntu 18.04).

for UDP, plus 20/40 bytes for IPv4/IPv6, plus 38 bytes for Ethernet). The packets from server to client occupied a total of 135 977 bytes of application-layer data. For comparison, a `mceliece6960119` public key by itself is 1 047 319 bytes.

## 10.5 CPU Usage

Haswell, introduced 2013, is not the newest Intel microarchitecture, but it is one of the most common optimization targets in recent papers on cryptographic primitives and in the NIST post-quantum project. SUPERCOP's latest benchmarks report the following speeds for `mceliece6960119` on one Haswell core: $0.71 \cdot 10^9$ cycles median for keygen (with high variance: the quartiles are $0.50 \cdot 10^9$ and $1.31 \cdot 10^9$), 153 944 cycles for enc (much less variance: quartiles 148 612 and 169 396), and 305 880 cycles for dec (quartiles 304 616 and 306 232).

We collected `mctiny6960119` timings on a Haswell, specifically a quad-core 3.1GHz Intel Xeon E3-1220 v3, for comparability to these microbenchmarks. To estimate the total server time, we ran a series of 1000 key exchanges and observed in `ps` that the `server` process had accumulated 17 seconds of CPU time, i.e., 17 milliseconds (53 million cycles on one CPU core) per key exchange. Generic packet processing can incur significant costs that the OS does not attribute to the process, but this measurement shows that the `mctiny6960119` server computations consumed only about 40 CPU cycles per byte communicated. (The client computations are more expensive since the client generates a short-term public key.)

We also instrumented the server with calls to a cycle counter, printing and resetting the counter whenever the server computed a session key. These figures showed that the server took 44.4 million cycles per key exchange (standard deviation 1.1 million cycles) for all activities outside `recvfrom` and `sendto`. Within the 44.4 million cycles, 20.8 million cycles (standard deviation 0.4 million cycles) were spent on the core cryptographic computations in phase 1: regenerating the low-weight vector from a seed and computing the corresponding partial encryption.

## 10.6 Security Against Server CPU Overload

An attacker trying to overload a quad-core 3.1GHz CPU with 10Mbps of network traffic needs to consume 10000 cycles per byte. (A site paying for a larger Internet connection can, presumably, afford more than one CPU to handle the load.) In our server software, the maximum cycles per byte are spent in McEliece decapsulation for the first packet, about 400 cycles per byte to handle 810 bytes of application-layer data (and slightly fewer cycles per byte when bandwidth overhead is taken into account).

Note that adding the encrypted 512-byte extension field to the first packet has the side effect of reducing the load per byte. For comparison, *unencrypted* zero-padding of query packets is well known to reduce query amplification and other per-query-byte costs, but this protection could be compromised by network links that try to compress packets.

## 10.7 Security Against Memory Flooding

At no time does the server allocate any per-client storage. Each client packet is handled immediately and then forgotten. We built the server software to avoid allocating memory in response to client packets; we audited the source code for this property; and we checked with `strace` that, once the program entered its packet-handling loop, its system calls consisted entirely of `recvfrom`, `sendto`, and an occasional[6] key-file access. In short, this is a tiny network server, making it immune to server-memory denial of service.

[6]The server automatically caches each key for 1 second, or 10000 uses, whichever comes first.

## 11 Conclusions and Further Considerations

The previous sections have shown that at very little overhead in the number of packets and a few extra round trips, the conservative McEliece system can be fit into tiny network servers for forward secrecy without using any per-client memory.

Server operators might be concerned about the generous usage of randomness on the server side. We point out that the random nonces can be generated by advancing a stream cipher. Server operators might also be concerned about the cost of hashing. We used hash to simplify the description. Any way of deterministically deriving subkeys from a master key works and is often cheaper.

The analysis of nonce reuse attacks took up a significant portion of the security analysis. Our choice of XSalsa20-Poly1305 already limits the potential for damage but designers could replace AE with a wide-block cipher to further limit this potential. Such ciphers are currently less common and we managed to achieve protection without this choice, but the analysis would be simpler.

We encourage further analysis, including proofs if possible, of McTiny and variants of McTiny.

## Acknowledgments

This work was supported by the U.S. National Science Foundation under grant 1913167, by the European Commission under Contract ICT-645622 PQCRYPTO, and CHIST-ERA USEIT (NWO project 651.002.004), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA-390781972, by the Netherlands Organisation for Scientific Research (NWO) under grant 628.001.028 (FASOR), and by the Cisco University Research Program.

## References

[1] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *EUROCRYPT (2)*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019. https://eprint.iacr.org/2019/089.

[2] Daniel Augot, Lejla Batina, Daniel J. Bernstein, Joppe Bos, Johannes Buchmann, Wouter Castryck, Orr Dunkelman, Tim Güneysu, Shay Gueron, Andreas Hülsing, Tanja Lange, Mohamed Saied Emam Mohamed, Christian Rechberger, Peter Schwabe, Nicolas Sendrier, Frederik Vercauteren, and Bo-Yin Yang. Initial recommendations of long-term secure post-quantum systems, 2015. PQCRYPTO project https://pqcrypto.eu.org/docs/initial-recommendations.pdf.

[3] Tuomas Aura and Pekka Nikander. Stateless connections. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11–14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 87–97. Springer, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.4436.

[4] Daniel J. Bernstein. SYN cookies, 1996. https://cr.yp.to/syncookies.html.

[5] Daniel J. Bernstein. DNSCurve: Usable security for DNS, 2009. https://dnscurve.org.

[6] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece. Submission to NIST post-quantum call for proposals, 2017. https://classic.mceliece.org/.

[7] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast Constant-Time Code-Based Cryptography. In *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013. https://binary.cr.yp.to/mcbits.html.

[8] Matt Braithwaite. Experimenting with post-quantum cryptography, 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[9] Robert E. Calem. New York's Panix service is crippled by hacker attack, 1996. https://partners.nytimes.com/library/cyber/week/0914panix.html.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *Communications of the ACM*, 60:58–66, 2017. https://queue.acm.org/detail.cfm?id=3022184.

[11] CDN Planet. Initcwnd settings of major CDN providers, 2017. https://www.cdnplanet.com/blog/initcwnd-settings-major-cdn-providers/.

[12] Eric Crockett, Christian Paquin, and Douglas Stebila. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. https://eprint.iacr.org/2019/858.

[13] Alexander W. Dent. A Designer's Guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding, 9th IMA International Conference, Cirencester, UK, December 16-18, 2003, Proceedings*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2003. https://eprint.iacr.org/2002/174.

[14] Jason Fairlane. Flood warning, 1996. https://archive.org/download/2600magazine/2600_13-2.pdf.

[15] Ryan Gallagher and Glenn Greenwald. How the NSA Plans to Infect 'Millions' of Computers with Malware, 2014. https://theintercept.com/2014/03/12/nsa-plans-infect-millions-computers-malware/.

[16] V. D. Goppa. A new class of linear correcting codes. *Problemy Peredači Informacii*, 6(3):24–30, 1970. http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=1748&option_lang=eng.

[17] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. https://eprint.iacr.org/2011/506.

[18] Kris Kwiatkowski. Towards post-quantum cryptography in TLS, 2019. https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/.

[19] Krzysztof Kwiatkowski, Nick Sullivan, Adam Langley, Dave Levin, and Alan Mislove. Measuring TLS key exchange with post-quantum KEM, 2019. Second PQC Standardization Conference, https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kwiatkowski-measuring-tls.pdf.

[20] Adam Langley. CECPQ2, 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html.

[21] Adam Langley. Post-quantum confidentiality for TLS, 2018. https://www.imperialviolet.org/2018/04/11/pqconftls.html.

[22] Marek Majkowski. Fixing an old hack - why we are bumping the IPv6 MTU, 2018. https://blog.cloudflare.com/increasing-ipv6-mtu/.

[23] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory, 1978. JPL DSN Progress Report http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[24] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. The case for moving congestion control out of the datapath. In Sujata Banerjee, Brad Karp, and Michael Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 101–107. ACM, 2017. https://people.csail.mit.edu/alizadeh/papers/ccp-hotnets17.pdf.

[25] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15:159–166, 1986. http://citeseerx.ist.psu.edu/showciting?cid=590478.

[26] NIST. Post-quantum cryptography: Round 1 submissions, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[27] NIST. Post-quantum cryptography: Round 2 submissions, 2019. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[28] Bill Nowicki. NFS: Network File System protocol specification, 1989. https://tools.ietf.org/html/rfc1094.

[29] Phillip Remaker. IPv6 MTU gotchas and other ICMP issues, 2011. https://blogs.cisco.com/enterprise/ipv6-mtu-gotchas-and-other-icmp-issues.

[30] Jim Roskind. QUIC: Quick UDP Internet connection, 2013. https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf.

[31] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. Trickles: A stateless network stack for improved scalability, resilience, and flexibility. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005. https://www.cs.cornell.edu/~ashieh/trickles/trickles-paper/trickles-nsdi.pdf.

[32] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. A stateless approach to connection-oriented protocols. *ACM Trans. Comput. Syst.*, 26(3), 2008. https://www.cs.cornell.edu/people/egs/papers/trickles-tocs.pdf.

[33] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997. https://arxiv.org/abs/quant-ph/9508027.

[34] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM Computer Communication Review*, 21:133–147, 1991.

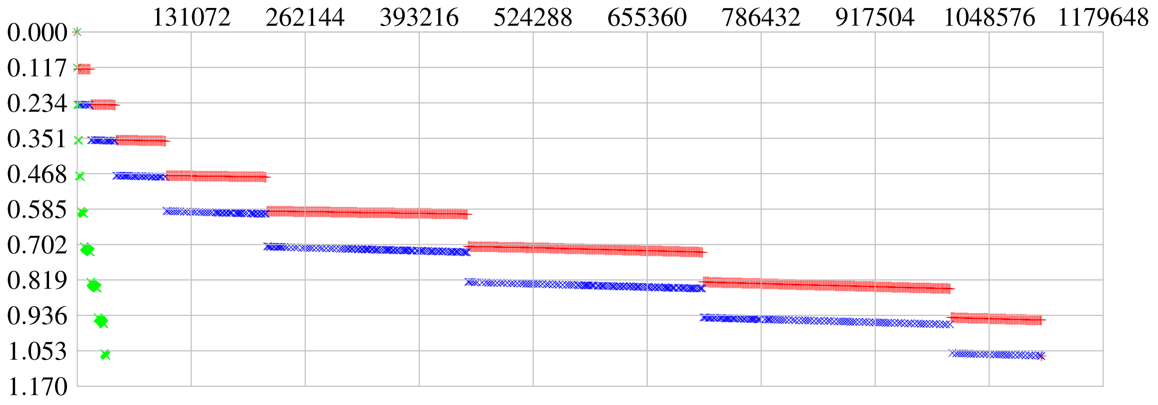


Figure 2: Timing of network packets observed by a server that accepts a TCP connection and sends 1MB.

## A Latency and Congestion Control

There are two obvious limits on the speed of a network protocol. There is also an unobvious limit, which is the main topic of this appendix.

As a running example, this appendix reports measurements of data transfer between one computer in the United States and another computer in Europe. The long-distance link between these two sites is reportedly able to handle 100Mbps, and the LANs can handle more than this. The minimum ping time we observed between the two computers is marginally under 0.117 seconds. The obvious limits are as follows:

- Each packet consumes bandwidth. This 100Mbps network connection cannot transmit more than 12.5 megabytes per second. Furthermore, not all of this data is application-layer data: as mentioned earlier, the total packet size is limited, and there are per-packet overheads.

- Sometimes a packet is in reply to a previous packet, and thus cannot be sent until that packet is received. The flow of data in a protocol implies that a certain number of round trips must be consumed, no matter how much bandwidth is available for sending packets in parallel.

To see that this is not the complete picture, consider a test TCP server that accepts a connection and then sends a server-specified amount of data over the connection. The second limit forces this connection to take at least two round trips, i.e., 0.234 seconds, and this is the latency we observed for small amounts of data. For 1 megabyte (more precisely, exactly $2^{20}$ bytes) we saw 1.066 seconds (average over 100 experiments, standard deviation 0.024 seconds), i.e., two round trips plus 0.832 seconds. Evidently only 1.25 megabytes per second were being transmitted during these 0.832 seconds.

One might try to explain this as the total 12.5-megabyte-per-second bandwidth being split across 10 users, so that each user has only 1.25 megabytes per second of available bandwidth. However, the network was not actually so heavily used. We measured sending 10 megabytes and saw 3.67 seconds (average over 100 experiments, standard deviation 0.46 seconds), more than 3 megabytes per second. Three experiments with sending 100 megabytes took 12.4 seconds, 17.8 seconds, and 19.1 seconds respectively, in each case more than 5 megabytes per second.

The reason that short TCP connections are slower—the unobvious limit mentioned above—is congestion control. We now briefly review the basic principles of congestion control, and then give an example of the exact timing of a McTiny connection using our implementation of congestion control.

### A.1 A Brief Introduction to Congestion

Suppose a router receives packets on a fast LAN more quickly than it can deliver those packets to the Internet. The packets pile up in a buffer inside the router; this is called **congestion**. A packet is not delivered until previous packets are delivered; the delay while a packet is waiting in the router's buffer is called **congestion delay**. If the buffer fills up then packets are lost; this is called **congestion loss** and produces further slowdowns. Routers often provide large buffers (**bufferbloat**) to try to avoid congestion loss, but these buffers allow congestion delay to increase even more.

TCP senders impose various limits upon their packet-sending rates to try to reduce congestion when there are signs of congestion, and to try to avoid creating congestion in the first place. This is called **congestion control**. The details are the topic of thirty years of active research.

In particular, when a TCP connection begins, the sender starts slowly, in case the network does not have much available bandwidth. For example, Akamai sends at most 32 packets at first; Cloudflare sends at most 10, which is also the current Linux default; see [11] for a broader survey. The sender then ramps up speed as long as acknowledgments show that the data is flowing smoothly—but acknowledgments arrive only after a round trip.

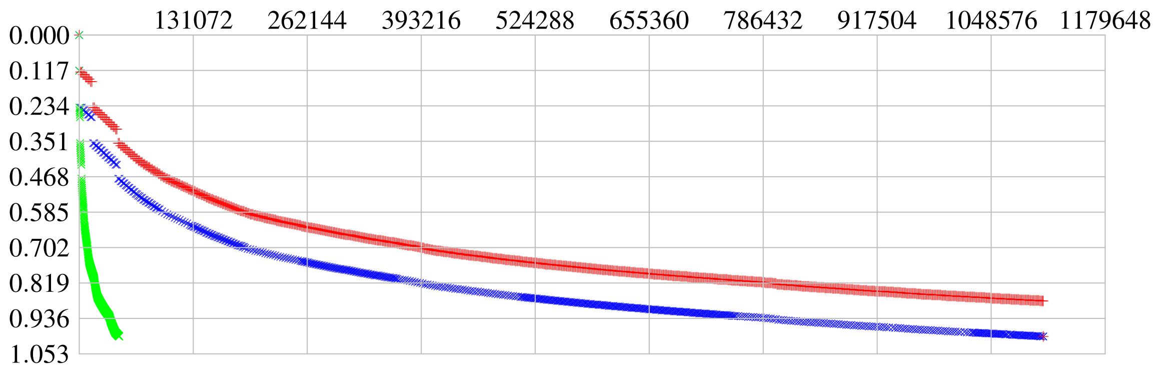


Figure 3: Similar to Figure 2, but telling the Linux TCP stack to use BBR instead of CUBIC.

## A.2 Measuring TCP Congestion Control

Figure 2 shows the timings of packets in a typical example of the experiments mentioned above, sending 1 megabyte through TCP from the United States to Europe. Each network packet sent by the server produces a red plus in the figure. The vertical position is the time in seconds when the server sends the packet. The horizontal position is the total number of bytes in all packets that have been sent, including the application-layer data (eventually reaching 1 megabyte) and 78 bytes of per-packet overhead: 20-byte TCP header, 20-byte IPv4 header, 26-byte Ethernet header, and 12 bytes of spacing between Ethernet packets.

Beware that packet-tracing tools such as `tcpdump` report the size of each Ethernet packet without the 12 bytes of spacing. Also, **TCP segmentation offload** means that the kernel gives larger packets to the network card, which then puts smaller packets on the wire; packet-tracing tools show the larger packets. We ran `ethtool --offload eth0 tx off rx off` to disable TCP segmentation offload, so that the same tools would show the packets on the wire; we did not find any resulting differences in TCP latency.

Each network packet received by the server produces a blue cross and a green cross in the figure, at the time in seconds when the server receives the packet. These packets acknowledge receipt of various packets sent earlier by the server. The horizontal position of the blue cross is the total number of bytes in the acknowledged packets, while the horizontal position of the green cross is the total number of bytes in the acknowledgment packets.

At time 0.000, the server receives a SYN packet opening a connection, and sends a SYNACK packet in response. At time 0.117, the server receives an ACK packet. After about 0.005 seconds of starting a data-generating application, the server sends a quick burst of 10 packets. Many more packets are ready to send, and could be sent given the available bandwidth, but the server is not yet confident about the available bandwidth.

These 10 packets are acknowledged slightly after time 0.234, prompting the server to send a burst of 20 packets. The burst size continues ramping up exponentially for a few more round trips. For example, there is a red burst of about 120000 bytes starting around time 0.468, creating about 0.010 seconds of congestion delay in the network router. These packets are delivered to the client at about 100Mbps, and the acknowledgments from the client create a blue burst in the figure starting around time 0.585 with a slope of about 100Mbps. This in turn triggers a longer red burst starting around time 0.585 with a slope of about 200Mbps, creating more congestion delay in the router. The difference between red and blue angles in the figure reflects the difference between 100Mbps and 200Mbps.

Overall this TCP server sent 769 packets, including 1 packet to accept the connection, 766 packets that each sent 1368 bytes of application-layer data (the maximum amount the client was willing to accept; note that this was over IPv4 rather than IPv6), 1 packet that sent the remaining 688 bytes of application-layer data, and 1 packet to acknowledge the client closing the connection (which this client did not do until after receiving all the server data). These packets consumed 1 108 566 bytes including per-packet overhead. Meanwhile the TCP client sent 430 packets, consuming 33 548 bytes including per-packet overhead. Note that TCP typically acknowledges two packets at once.

Figure 2 used CUBIC, the default congestion-control mechanism in Linux. Figure 3 instead uses BBR [10], a new congestion-control mechanism from Google; `sysctl net.core.default_qdisc=fq` followed by `sysctl net.ipv4.tcp_congestion_control=bbr` enables BBR under Linux. There are many differences between CUBIC and BBR, and one of these differences is already visible just after time 0.117: instead of sending a burst of 10 packets as quickly as possible, the server sends 5 separated bursts of 2 packets each. This separation ("packet pacing" from [34]) reduces the chance of producing immediate congestion, and in general produces a smoother data flow. Comparing the figures also shows that BBR sent slightly more acknowledgment traffic (590 packets from the client,

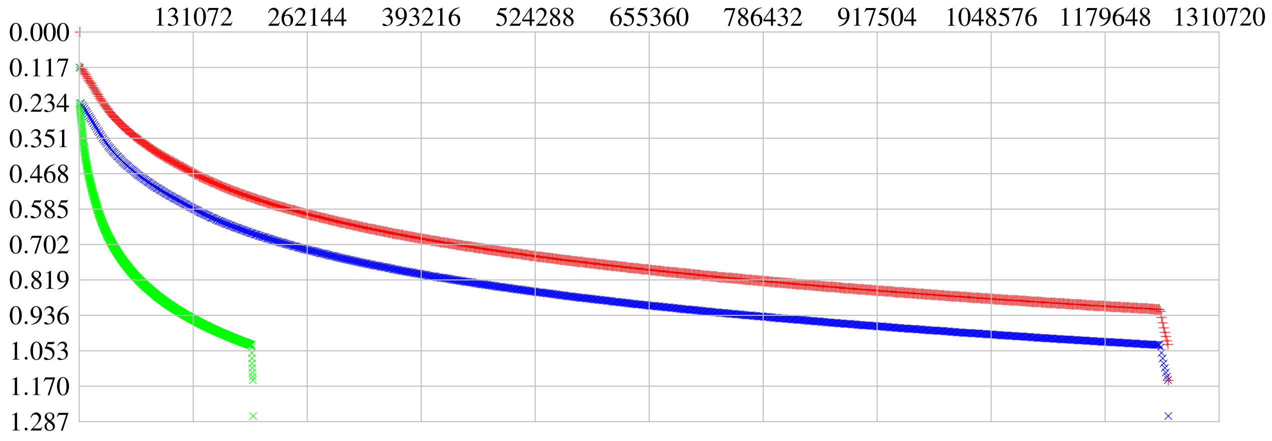


Figure 4: Timing of network packets observed by a McTiny client.

consuming 46 028 bytes including per-packet overhead) than CUBIC did, and also that BBR sent more data between time 0.702 and time 0.819 than CUBIC did, saving time overall.

The bottom line is that, because of congestion control, TCP takes about 9.1 round-trip times to send 1MB using CUBIC, or 8.5 round-trip times to send 1MB using BBR. Smaller congestion-control details also affect the latency: e.g., raising the initial packet limit from 10 to 32 would have saved more than 1 round-trip time.

## A.3 Building McTiny Congestion Control

We decided to integrate TCP-like congestion control into our McTiny software. TCP itself is incompatible with the concept of a tiny network server, but, as mentioned earlier, congestion control can be managed entirely by the client.

There is a software-engineering problem here. Congestion-control software is typically developed as part of a monolithic TCP network stack, and interacts with the rest of the network stack through a thick interface, so reusing the software outside the TCP context is difficult. There have been efforts to build reliable network protocols on top of UDP, and some of these protocols—e.g., Google's QUIC [30]—imitate TCP's congestion-control mechanisms, but again we did not find something easy to reuse.

We thus wrote yet another implementation of congestion control. We put some effort into designing a simple interface for future reusability, taking some API ideas from [24] but building a userspace library rather than a tool designed to integrate with the OS kernel. We first implemented CUBIC but found that the bursts of traffic in CUBIC frequently overload UDP buffers (which are typically configured by the OS with less space than TCP buffers), creating packet losses and often considerable slowdowns. We considered variants of CUBIC with pacing but in the end threw CUBIC away and implemented BBR. As explained in [10], BBR handles packet loss much better than CUBIC, and tends to avoid overloading buffers in the first place.

## A.4 Measuring McTiny Congestion Control

Figure 4 shows an example of the timing of all of the network packets in one McTiny run between the computer in the United States and the computer in Europe. The CPUs on these computers were, respectively, a quad-core 3.1GHz Intel Xeon E3-1220 v3 (Haswell) and a quad-core 3.5GHz Intel Xeon E3-1275 v3 (Haswell). The elapsed `client` time measured by `time` was 1.664 seconds, including 0.423 seconds of "user" CPU time (on a single core; this is about 6% of the time available on a quad-core CPU in 1.664 seconds of real time) and 0.009 seconds of "sys" CPU time. Most of the CPU time is for generating an ephemeral McEliece key, which the client could have done any time in advance.

The total vertical spacing in the figure covers 1.268 seconds, about 10.9 round-trip times. Each packet is shown at the time it is sent or received by the client. For comparison, Figures 2 and 3 show times on the server, but in those cases the 1MB of data was being sent by the server whereas in Figure 4 the 1MB of data is being sent by the client.

As the figure shows, our BBR implementation paces packets somewhat more smoothly than the Linux TCP BBR implementation, but overall we increase rate along essentially the same curve as in Figure 3. The last few round trips in McTiny transmit much less data; the red, blue, and green curves are close to vertical at this point. There is more data sent and received in Figure 4 than in Figure 3—there is more overhead in each packet for cryptographic protection, data is sent in somewhat smaller packets, and each packet is acknowledged—but this makes relatively little difference in latency.

To summarize, our McTiny software is using the network in this example with similar efficiency to TCP, plus two round-trip times for final cleanup in the McTiny protocol. For our software, as for TCP, the first megabyte of data sent through this network is limited primarily by congestion control.