# *Open Questions in Encryption*

*6.1600 Course Staff*

*Fall 2023*

So far, we have established what may seem like a comprehensive set of tools to transmit data over the network: we have schemes for verifying the integrity of data and for hiding the contents of a transmission from an adversary, both with and without a shared key.

Transport-layer security (TLS) effectively builds and "encrypted pipe" between a client and a server. Through the encrypted pipe that TLS provides, we can run any of our favorite TCP-based protocols—HTTP, SMTP, POP, IMAP, etc.—and can thus hide our data from an in-network attacker. And yet, the security guarantees that TLS provides fall short of the strongest possible security notions we could desire.

If we were to imagine the best possible security we could ask for regarding network traffic, it might look (imprecisely) something like the following:

> *An attacker who controls many parties (clients and servers) as well as the network should "learn nothing" about who the client is talking to and what she is saying.*

Unfortunately, the protocols we have for secure communication today fall far short of this goal. In this section, we describe some of the shortcomings of today's transport-security tools and some imperfect solutions.

**Disclaimer:** This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

## *1 Problem: Encryption does not hide the source and destination of a packet*

In order to send IP packets over the internet, the Internet's routing system relies on routers in the network knowing the source and destination IP addresses of each packet: these are included, unencrypted, in the packet header. (In some ways, the "pipe" analogy fits here: anyone can see where the pipe starts and where it ends.)

*Solution Attempt: Tor* The Tor system aims to allow a client to connect to a server over the Internet while hiding—from certain types of adversaries—which server the client is connecting to. For example, a Tor client should be able to browse the web without anyone learning which websites the client is visiting.

Tor's strategy is to bounce traffic around the Internet and hope that no real-world attacker can gather enough information to figure out which client is communicate with which server. Tor provides no precise security guarantees, and there are scores of research papers demonstrating various weaknesses in Tor's security plan. At the same time, Tor is publicly available, is well supported, is widely used, and seems to provide some meaningful privacy benefits in practice.

It is difficult to evaluate the security of a tool like Tor, since real-world attackers will not necessarily reveal that they can break the tool's security guarantees. So using a tool like Tor requires taking a leap of faith.

Tor works by nesting several of these encrypted pipes: when opening a connection, the Tor client will first select three *relays* from the Tor network $(A, B, C)$. The Tor client will then:

1. Open an encrypted tunnel to the first relay $A$ (the "guard").
2. Through that tunnel, open an encrypted tunnel to the second relay $B$.
3. Through that tunnel-inside-a-tunnel, open an encrypted tunnel to the third relay $C$ (the "exit").

The client will then send its application-layer traffic through this tunnel-inside-a-tunnel-inside-a-tunnel. So each byte of application data will be encrypted first for relay $C$, then for relay $B$, then for relay $A$. When the client sends this ciphertext over the *circuit* from relay $A$ to $B$ to $C$ to the real destination, relay $A$ will first strip off its layer of encryption then forward the inner packet to relay $B$. Relay $B$ will do the same, stripping off a layer of encryption and forwarding the packet to relay $C$. Finally, relay $C$ will strip off the last layer of encryption and be left with a normal IP packet that it can then send to the destination server. As the response makes it back through the network, each relay node will add a layer of encryption. The end result of this is that no single relay can see the source and destination IP addresses.

However, the security that Tor provides is imperfect. First, if an attacker controls the guard node (relay $A$) and the exit node (relay $C$), the attacker can correlate the timing of when a packet enters the guard node and when a packet exits the exit node. Using this timing an attacker can make a guess at the route traffic is taking through the Tor network. Even without controlling relay nodes, if an attacker controls certain key points in the Internet (e.g., Internet exchange points or undersea fiber links) it may be able to perform this sort of traffic analysis even without controlling relays.

## 2 *Problem: Attacker sees packet sizes and timings*

As we discussed, practical encryption schemes necessarily reveal the length of the ciphertext. In the context of the Internet, this means that

an attacker can learn the size of each TCP packet that a client sends, along with timing information. (Here the "encrypted pipe" analogy for TLS breaks down: an attacker can see how much traffic flows through the encrypted pipe and when.)

Even without seeing the destination and source of packets sent to and from a client's machine, an attacker can learn significant amounts about the client's traffic. Some examples are:

- *Watching a movie:* Video traffic has a distinct traffic pattern. By monitoring, for example, the length of time that a client spends watching a movie, a network attacker learn with fairly high accuracy *which movie* the client is watching.
- *Using* `ssh`*:* Different commands will have different traffic patterns. An attacker may be able to infer what type of commands a client is running by inspecting traffic patterns.
- *Downloading a file:* The bitlength of a downloaded file can uniquely identify the file in many cases.
- *Browsing the web:* sizes leak individual pages.

*Example: New York Times* The New York Times homepage `nytimes.com/` downloads 1.56 MB of content, along with 76 total assets (images, CSS, JavaScript). The webpage to submit a sensitive tip, `nytimes.com/tips`, downloads 41.92 KB and only 15 assets. By counting the number of HTTPS requests that a client makes over an encrypted connection to `nytimes.com`, an attacker can easily distinguish whether a client is visiting the homepage or the tips page, even if the attacker cannot decrypt even a single bit of the HTTPS traffic itself.

*Attempts at a solution* There are several common ways that people attempt to protect against this sort of traffic analysis. None of these solutions works well.

1. **Random Noise:** To try to hide the length of the packets it sends, a client can add a randomly chosen number of bytes of dummy data to the end of each packet. The hope is that by randomizing packet lengths, the client prevents the attacker from performing the traffic analysis.

   Unfortunately, a patient attacker can use *averaging* to effectively eliminate the effect of the random noise. That is, if the attacker can trick the client into sending the same message a few times (as is often possible), the attacker can average the noised packet lengths to get a good estimate of the true length.

2. **Padding:** Another option is to just pad every packet (or webpage or encrypted message, etc.) to match the largest packet that the client will ever send. For example, whenever the client visits a page on `nytimes.com`, the client could download 50MB of page content and 100 fixed-size assets, even if the true page is tiny. This is somewhat secure, but incredibly costly and therefore not practical outside of very specific circumstances.

## 3 *A Promising Direction: Metadata Privacy for Messaging*

Messaging apps like WhatsApp and iMessage are end-to-end encrypted, but still may leak who you are talking to. This problem is more tractable due to the circumstances of messaging:

- Messages are approximately fixed length.
- Some latency is OK.
- Total daily traffic per user is small.
- Each user talks to few messaging partners.

Because of these constraints, it may be feasible to use techniques like padding to greatly reduce the amount of data that messaging metadata reveals and to do so in a way that provides strong formal guarantees about security. But still, no widely used messaging app provides any sort of metadata-privacy guarantees.

## 4 *Problem: Endpoint Compromise*

Say that we have a perfect scheme for transport security—one that hides all data and metadata. Such a scheme is still not enough to protect our data if an adversary can compromise the communication *endpoints*.

For example, in many applications, a client sends some sensitive data to a server (e.g., its Google search queries). The server is free to lose it in a breach, sell it, or turn it over to law enforcement agencies, etc. Later on, we will discuss how to protect against server compromise using software-engineering techniques. Here, we will give one example of how cryptography can protect user data even against a compromised server.

### 4.1 *Private Information Retrieval*

In many applications, a client must read a record from a database stored at a server. The client might like to perform such a database query without the server learning which record it accessed.

A concrete application of this is Google search—in order to give you search results, Google necessarily learns what you are searching. With a PIR scheme, it would be possible for Google to look up search results without learning what you are searching for!

In a *private information retrieval* scheme:

- the server holds a public database of $n$ bits: $x_1, x_2, \ldots, x_n \in \{0,1\}$, and
- the client holds a secret index $i \in \{1, \ldots, n\}$.

The client and server interact. At the end of the interaction we want the following properties to hold:

- *Correctness:* The client outputs $x_i \in \{0,1\}$.
- *Security:* The server "learns nothing" about the client's secret index $i$. In particular, we demand that the message that the client sends to the server is a CPA-secure encryption of its index $i$.

*Naïve private information retrieval.* The simple scheme for private information retrieval is to have the server send all $n$ bits of the database to the client. When the database is large, as it is for Google search, this would be an infeasible amount of communication. A surprising fact is that there are simple private-information-retrieval protocols that involve much less than $n$ bits of client-server communication.

### 4.2 *A non-trivial private-information-retrieval scheme.*

To achieve this, we need a new tool called *additively homomorphic encryption*.

*Additively homomorphic encryption* A secret-key additively homomorphic encryption scheme is a CPA-secure secret-key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ over key space $\mathcal{K}$ and message space in $\mathcal{M} = \mathbb{Z}_p$ with the added property that for all keys $k \in \mathcal{K}$ and all messages $m, \hat{m} \in \mathcal{M}$,

$$\mathsf{Enc}(k, m) \star \mathsf{Enc}(k, \hat{m}) = \mathsf{Enc}(k, m + \hat{m}),$$

where "$\star$" is some fixed binary operation on ciphertexts.

In English: given two encrypted messages $m$ and $\hat{m}$, encrypted under an additively homomorphic encryption scheme, anyone can compute the encryption of $m + \hat{m}$. Being able to add encrypted messages also allows multiplying encrypted messages by public constants, since $m + m = 2m$ and $2m + 2m = 4m$ and so on. Once we can add and multiply by constants, and we can compute a matrix-vector product of an encrypted vector and a public matrix.

It is possible to construct an additively homomorphic encryption scheme from the DDH assumption, with only a slight tweak to ElGamal encryption.

*PIR Construction* We can use additively homomorphic encryption to construct a private-information-retrieval scheme. To do so, the server represents its its database (the $x_i$ values) as a $\sqrt{n} \times \sqrt{n}$ matrix $D$. The client then tells the server which column $j$ it would like by supplying the encryption of a $\sqrt{n} \times 1$ vector $\mathbf{m}$ with a 1 in the $j$th location. The client sends this encryption (using a key only the client knows) as $\mathsf{Enc}(k, \mathbf{m})$. The server computes the matrix product $\mathsf{Enc}(k, D\mathbf{m})$, which gives the $j$th column of the matrix, using additively homomorphic encryption and returns the response to the client, where the client can find the bit they are interested in in the column.

This allows a client to retrieve a bit from a server's database without the server learning anything about the desired bit, and to do so at the communication cost of only $2\sqrt{n}$ ciphertext. The server computation cost is high—the server necessarily touches every bit of the database—but at least it shows that making private queries is feasible in theory.