View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCL Discovery

The case for ubiquitous transport-level encryption

Andrea Bittau, *Stanford*
Michael Hamburg, *Stanford*
Mark Handley, *UCL*
David Mazières, *Stanford*
Dan Boneh, *Stanford*

Abstract

Today, Internet traffic is encrypted only when deemed necessary. Yet modern CPUs could feasibly encrypt most traffic. Moreover, the cost of doing so will only drop over time. *Tcpcrypt* is a TCP extension designed to make end-to-end encryption of TCP traffic the *default*, not the exception. To facilitate adoption tcpcrypt provides backwards compatibility with legacy TCP stacks and middleboxes. Because it is implemented in the transport layer, it protects legacy applications. However, it also provides a hook for integration with application-layer authentication, largely obviating the need for applications to encrypt their own network traffic and minimizing the need for duplication of functionality. Finally, tcpcrypt minimizes the cost of key negotiation on servers; a server using tcpcrypt can accept connections at 36 times the rate achieved using SSL.

1 Introduction

Why is the vast majority of traffic on the Internet not encrypted end-to-end? The potential benefits to end-users are obvious—improved privacy, reduced risk of sensitive information leaking, and greatly reduced ability by oppressive regimes or rogue ISPs to monitor all traffic without being detected. In spite of this, end-to-end encryption is generally used only when deemed *necessary*, a small fraction of when it would be *feasible*.

Possible reasons for not encrypting traffic[1] include:

- Users don't care.
- Configuration is complicated and the payoff small (especially when connecting to unknown sites).
- Application writers have no motivation.
- Encryption (and key bootstrap) are too expensive to perform for all but critical traffic.
- The standard protocol solutions are a poor match for the problem.

We believe that each of these points either is not true, or can be directly addressed with well-established techniques. For instance, where users actually have control, they demonstrate that they do care about encryption. Four years ago only around half of WiFi basestations used any form of encryption [3]. Today it is rare to find an open basestation, other than ones which charge for Internet access.

It is clear, though, that application writers have little motivation: encryption rarely makes a difference to whether an application succeeds. Getting it right is difficult and time consuming, doesn't help time to market, and developers are hard-pressed to make the business case. For server operators, too, the process can be tedious. One reason people don't use SSL is that X.509 certificates are a mild pain both for the server administrator and, if the server administrator didn't buy a certificate from a well-known root CA, for users.

Even more important is the performance question. SSL is by far the most commonly deployed cryptographic solution, and it is expensive to deploy on servers. Where there is a need, such as for bank login or credit card payments, SSL is ubiquitous, but it is rarely used outside of web pages that are especially sensitive. The definition of "sensitive" has started to change, though; Google recently enabled SSL on all Gmail connections [25], ostensibly as a response to eavesdropping in China. In part this is possible today because cryptographic hardware has become comparatively inexpensive. This trend is set to continue; the most recent generation of Intel CPUs incorporate AES acceleration instructions [8], with the potential to significantly reduce the cost of software symmetric-key encryption.

Although symmetric-key encryption is unlikely to be

[1] Conspiracy theorists might suggest other reasons, but we won't discuss those here.

a problem, the conventional wisdom is still that it is too expensive to use public-key cryptography to bootstrap a session key for all network connections. Indeed our measurements show that a fully loaded eight-core (2 x Quad-core Xeon X5355) server can only establish 754 uncached SSL connections per second. In fact, this limitation is due to the way SSL uses public key algorithms rather than anything fundamental. We will show that much better server performance is possible with the right protocol design, in part by pushing costs to the client, which does not need to handle high connection rates.

Finally, there is the question of whether current encryption protocols are a sufficiently good match for applications that do not currently use encryption. We believe they are not, for reasons we shall highlight throughout the paper. However, we will describe a subtly different protocol architecture that we believe is a much better fit to the majority of applications. This is not rocket science; it may even be considered obvious. But we believe it makes a huge difference to the deployability of encryption and consequently of authentication in the real world.

1.1 Getting the Architecture Right

All the commonly deployed network encryption mechanisms incorporate authentication into the protocol, even if, like WPA, it is as simple as requiring out-of-band password exchange. Indeed this is the obvious way to engineer things; without authentication, it is not possible to determine if your encrypted channel is with the desired party or with a man-in-the-middle. However, we believe that this is fundamentally the wrong design choice.

Encryption of a network connection is a general purpose primitive; regardless of the application, the goal is to prevent eavesdroppers from learning the contents of communications. MACing of packets in a network connection is also a general purpose primitive; no application wants to accept forged or maliciously modified packets. Authentication, however, is not general purpose. The mechanism used for authentication and the information needed to perform that authentication are application-specific. In practice, protocols blur this distinction between general purpose encryption/integrity and special purpose authentication. This has two consequences:

- It tends to encourage inappropriate authentication mechanisms. For example, using SSL to connect to a bank, then simply handing the user's password to the bank, when it is known that people commonly re-use passwords across sites.

- It makes it hard to integrate mechanisms low enough in the protocol stack to really be ubiquitous. For example, adding SSL to an application requires modifying the source code and, potentially, extending its application-layer protocol in a backwards compatible way.

To enable encryption and integrity checking in a general way for all legacy TCP applications[2], this functionality must be *below* the application layer. However it cannot be done *cleanly* any lower than the transport layer because this is the lowest place in the stack that has any concept of a conversation. There is also the practical consideration that encrypting below the transport layer will prevent NAT traversal. The clear implication is that embedding encryption and integrity protection *into* TCP would provide the right general-purpose mechanism; in fact, because TCP includes a session establishment handshake, this is simple to do in a backward-compatible way.

To establish session keys in a general way, TCP-level encryption should be divorced from higher level authentication mechanisms. This suggests the use of ephemeral public keys to establish session keys. Such a mechanism, enabled by default, would provide protection against passive eavesdroppers for all TCP sessions, even for legacy applications. We are not the first to suggest such "opportunistic" encryption. Our goal, though, is to provide not just encryption and integrity protection, but also a firm foundation upon which higher-level authentication mechanisms can build. With the right architecture, a diverse set of authentication mechanisms can be devised, each suitable to its own application.

The end point we hope to establish is that all TCP sessions (and SCTP and DCCP, though we don't discuss these further here) are protected against passive eavesdroppers, and that all applications that require authentication should, as a side effect, enjoy protection against active man-in-the-middle attacks, all without duplication of effort. Ideally, an eavesdropper cannot tell from watching the traffic which encrypted sessions will be authenticated.

In this paper, we describe *tcpcrypt*, our implementation of TCP-level encryption. Although the idea is simple, the details really matter, as we will show. We have validated our design by building two implementations, one a Linux kernel module, the other a user-space process using divert sockets. The latter allows use of tcpcrypt on Linux, FreeBSD, and MacOS X without modifying the kernel. Both implementations show excellent performance; we will demonstrate that this is no longer the factor preventing ubiquitous network encryption. We have also implemented application-level authentication protocols that use tcpcrypt to bootstrap authentication. These include X.509 certificate-based authentication, fast password-based *mutual* authentication, and PAKE. Our X.509-based authentication pro-

[2] The vast majority of Internet applications use TCP.

vides security equivalent to SSL, but uses batch-signing to run 25 times faster. Moreover, we have implemented X.509 authentication inside the OpenSSL library in a way that preserves the same API and cleanly falls back to vanilla SSL when appropriate. Thus, to take advantage of tcpcrypt in SSL-enabled applications requires only a library update.

2 Cryptographic design

The goal of tcpcrypt is to enable the best communications security possible under a wide range of circumstances. In the absence of any authentication, when users browse unknown servers, they should enjoy protection from passive eavesdropping. Though active network attackers may still intercept and monitor communications (there are also legitimate reasons for this, such as transparent proxies and intrusion detection systems), it should be possible to detect such behavior both during communications and afterward. Thus, tcpcrypt should virtually eliminate the possibility of widespread eavesdropping unbeknownst to a user population.

When an application performs any kind of endpoint authentication, it must be able to leverage tcpcrypt to obtain stronger protection of session data. For instance, given a server-side X.509 certificate, the client should be assured of the confidentiality of the data it transmits and the integrity of the data it receives. Any time a user types a password, it should be possible to ensure the confidentiality and integrity of all data sent in either direction.

In all cases, when tcpcrypt achieves confidentiality, it should also provide forward secrecy. As a final goal, tcpcrypt should affect performance as little as possible. Thus, the protocol is designed to minimize the number of cryptographic operations and extra round trips, subject to the limitations of needing to interoperate with legacy end hosts and middleboxes.

2.1 Key exchange protocol

Key exchange is the biggest challenge to tcpcrypt's performance. Forward secrecy requires a pair of hosts to exchange a secret using an ephemeral public key or Diffie-Hellman key exchange the first time they communicate. These operations are far more costly than establishing a TCP connection, but the cost can be asymmetric. For example, a single core of the server in Section 6 can perform 12,243 encryptions/sec with a 2,048-bit RSA-3 key, but only 97 decryptions/sec.

Servers typically communicate with more peers than clients do, so it makes sense for clients to shoulder most of the cost of key exchange. Thus, by default, tcpcrypt performs the expensive decryption at the client (though for generality, servers may opt to reverse the protocol).

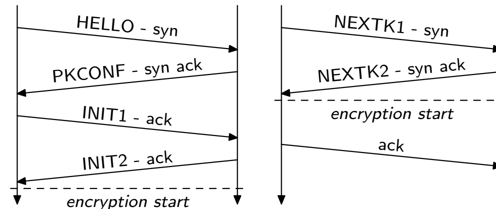


Figure 1: Tcpcrypt connection establishment with key exchange (left) and session caching (right).

Subsequent connections between the same two hosts can use *session caching* to avoid any public key operations at all, thereby ensuring that, for instance, an active-mode FTP server need not perform RSA decryptions.

The initial key exchange works as follows. Each machine C has an ephemeral public key, K_C. When C connects to a server S for the first time, C chooses a random nonce, N_C; S chooses a random secret, N_S; the two exchange the following messages, also shown in Figure 1:

$C \to S$: HELLO
$S \to C$: PKCONF, pub-cipher-list, [cookie]
$C \to S$: INIT1, sym-cipher-list, N_C, K_C, [cookie]
$S \to C$: INIT2, sym-cipher, ENCRYPT (K_C, N_S)

Here pub- and sym-cipher-list are used to negotiate cryptographic algorithms. The optional cookie is a SYN-cookie that must be echoed by the client to make it harder for packets from forged source addresses to trigger any public-key cryptographic operations in the server. This trade-off is at the discretion of the server; if TCP's 32-bit initial sequence number (ISN) provides enough protection against forged packets, the option space may be deemed better used for other purposes.

K_C specifies the public key cipher and a pseudo-random function, used below. Quantities from this protocol are then combined into a series of "session secrets" with a Collision-resistant Pseudo-random Function, CPF (currently HMAC):

$$\mathrm{ss}[0] \leftarrow \mathrm{CPF}\big(N_S, \{K_C, N_C, \text{cipher-lists}, \text{sym-cipher}\}\big)$$

$$\mathrm{ss}[i] \leftarrow \mathrm{CPF}\left(\mathrm{ss}[i-1], \texttt{TAG_NEXT_KEY}\right)$$

If $\mathrm{ISN}_{C,i}$ and $\mathrm{ISN}_{S,i}$ are TCP's initial sequence numbers on the client and server for session i, the two sides then compute a *master secret* as follows:

$$\mathrm{mk}[i] \leftarrow \mathrm{CPF}\left(\mathrm{ss}[i], \{\texttt{TAG_KEY}, \mathrm{ISN}_{C,i}, \mathrm{ISN}_{S,i}\}\right).$$

Finally, the two sides use $\mathrm{CPF}(\mathrm{mk}[i], x)$ on various constants x to generate encryption and MAC keys (a common technique). From this point on, all further segments in the TCP connection are cryptographically protected.

Note that this full key exchange is only needed for the first connection between two hosts. Hosts can cache $\mathrm{ss}[i]$

for the largest i used till that point. Subsequent connections between the same two hosts can use this to derive new symmetric keys, thereby avoiding any further public key cryptography and the latency of the full handshake.

2.2 Authentication Hooks

To gain stronger benefits from tcpcrypt, applications must be able to make statements about a connection—*e.g.,* "All data you read from this connection is sent by user U's browser," or "Any data you write to this connection can be decrypted only by server Y." To make such statements, one must specify what is meant by "this connection" in a way that cannot be interpreted out of context. Tcpcrypt accomplishes this through *session IDs*. A new *getsockopt* call returns a session ID, $\mathrm{sid}[i]$, computed from the connection's session secret $\mathrm{ss}[i]$ as follows:

$$\mathrm{sid}[i] \leftarrow \textsc{cpf}\,(\mathrm{ss}[i], \texttt{TAG_SESSION_ID})$$

If both ends of a tcpcrypt connection see the same session ID, then with overwhelming probability an attacker cannot eavesdrop on or undetectably tamper with traffic—*i.e.,* there has not been a man-in-the-middle attack. Two properties facilitate verification of session IDs. First, they need not be kept secret. Second, with overwhelming probability they are unique over all time, even if one end of a connection is malicious. Hence, a cryptographically endorsed session ID can only ever authenticate a single tcpcrypt connection. In Section 4 we discuss different ways applications can leverage session IDs.

2.3 Proof of Security

To increase confidence in tcpcrypt, we provide a semi-formal proof of its security. We assume that the adversary has complete control over the network, and nearly complete control over the users. It can choose when and to whom users attempt to connect, and what data they send, and can delay, drop, modify, and forge packets arbitrarily. Furthermore, since the session IDs $\mathrm{sid}[i]$ are not secret, we assume that the adversary knows them. We do not model malicious machines here, as the adversary can emulate as many of these as it wants. We do not model compromised machines because of space constraints. When we write "client" or "server" in this discussion, we mean a legitimate client or server.

We guarantee the security of tcpcrypt connections only when the session IDs match. In this case, the guarantee is fairly strong:

Definition 2.1 (Security guarantees). *Suppose that users U_1 and U_2 complete the tcpcrypt protocol on sockets S_1 and S_2, and arrive at sessions with the same session ID. Then the following guarantees hold:*

- *The adversary has not tampered with U_1 and U_2's cipher suite choices. Assuming they have chosen a secure cipher suite:*
- *Any packet sent by U_1 on socket S_1 (or by U_2 on S_2) gives no information to the adversary other than its length and timing.*
- *If, after TCP reassembly, U_2 receives a sequence of segments $p_1, \ldots, p_n$, then U_1 sent those segments in that order (and no segments before them), and similarly for segments received by U_1.*

We will show that, unless the adversary has broken the underlying cryptographic primitives, its probability of violating this guarantee is very small. Specifically:

Theorem 2.1 (Security of tcpcrypt). *Suppose that an adversary $\mathcal{A}$ can violate the tcpcrypt security guarantee with probability ϵ. Suppose that it uses m machines in its attack, and begins at most c connections in total. Then there are five simple modifications of $\mathcal{A}$, running in about the same time as $\mathcal{A}$, which aim to do the following things:*

- *Find a collision in* CPF.
- *Break the pseudorandomness of* CPF.
- *Break the public-key cipher.*
- *Break the MAC.*
- *Break the symmetric cipher.*

The sum of their probabilities of success is at least

$$\epsilon - 3c^2/2^{k+1}$$

where $k \approx 256$ is the minimum of the min-entropy of a public key, or the length in bits of N_S or N_C.

Proof. Define $\textsc{next}(k) := \textsc{cpf}(k, \texttt{TAG_NEXT_KEY})$. Suppose that U_1 and U_2 have the same sid, and that for U_1 it is $\mathrm{sid}[i]$ for some i, where:

$$\mathrm{ss}[0] = \textsc{cpf}\,(N_S, \{K_C, N_C, \text{cipher-lists, sym-cipher}\})$$
$$\mathrm{sid}[i] = \textsc{cpf}\,\left(\textsc{next}^i(\mathrm{ss}[0]), \texttt{TAG_SESSION_ID}\right)$$

Because everything passed to CPF has a unique parse, the sid must have been computed by U_2 in the same way—and in particular with the same values of N_S, N_C, K_C, the same cipher suite lists and the same cipher choice—or else the computation contains a hash collision. What is more, the N_S, N_C, and K_C values are chosen at random, and so with probability at least $1 - 3c^2/2^{k+1}$ they are unique. For the rest of the proof, assume that this is the case.

Now, each of U_1 and U_2 is either a client or a server. Because their K_C, N_C and N_S values match, they can't both be clients or both be servers; without loss of generality, say U_1 is the client (which generated K_C and N_C), and U_2 is the server (which generated N_S).

We will next show that this N_S remains secret. We first replace $\text{ENCRYPT}(K_C, N_S)$ with an encryption of zero (but the client still decrypts it to N_S). If the adversary notices this, then it has broken the public-key cipher. After this change, N_S is only used as a key to CPF. Furthermore, CPF is evaluated on N_S only once by U_2 and once by U_1, with a nonce N_C in the other argument; if the adversary replays $\text{ENCRYPT}(K_C, N_S)$, then $\text{CPF}(N_S, \cdot)$ will be called with different nonces. Because CPF is pseudorandom, we can replace its outputs ss[0] with independent random values; if the adversary notices this, then it has broken CPF. Continuing in this manner, we can replace ss$[i]$, mk$[i]$, sid$[i]$ and the encryption and MAC keys with random values, and the adversary will not notice this, either.

If the initial sequence numbers do not match, the client and server will arrive at different (secret, random) MAC keys, and so as long as the MAC is unforgeable, neither will accept any packets at all. Otherwise since every packet is MACed with associated data that includes the 64-bit extended sequence number, they must be received unmodified and in order. Finally, if the symmetric cipher is secure against chosen-plaintext attacks, the only information that the adversary can learn about a segment is its length and timing. This completes the proof. □

3 Integration with TCP

Integrating tcpcrypt into TCP posed a number of challenges ranging from the basic to the baroque. First, we have to extend TCP in a backwards compatible way. If a tcpcrypt client connects to a tcpcrypt server, encryption should be enabled by default, but if it is a legacy server, the session must fall back to regular TCP behavior.

The same issue applies with middleboxes. Tcpcrypt must work through NATs, so it cannot protect the TCP ports. Tcpcrypt must also work correctly when faced with firewalls that do not understand the tcpcrypt extensions. For an example of how broken firewalls have inhibited innovation, we need look no further than Explicit Congestion Notification (ECN). ECN should be harmless to deploy—it uses TCP options in the handshake to negotiate the capability, then uses two bits from the old IP Type-of-Service field to indicate congestion, and finally signals this in feedback using a previously reserved TCP flag. ECN is built into all the main modern operating systems, but is disabled by default. This is because a small number of home gateway/firewall boxes crash when they see the reserved TCP flag set to one.

This has taught us to avoid protocol changes to TCP that are not carried in TCP options. Firewalls might drop unknown options, or might completely drop packets with unknown extensions; a TCP extension needs to be robust to either and correctly fall back to regular TCP behavior.

Finally we risk being hoisted by our own petard. Traffic normalizers [9], as implemented in `pf` [10] and some other firewalls, enforce conservative rules on protocol behavior and consistency. This limits design flexibility.[3]

3.1 Initial TCP Handshake

Ideally the key exchange for tcpcrypt would be performed in TCP's three-way connection setup handshake, as this would add no additional network latency to establishing encrypted sessions. We can't quite achieve this for the first connection between two hosts—rather, we require adding information to the first four packets of the session, as shown in Figure 1. To be backwards compatible with regular TCP, any data we can add to the SYN and SYN/ACK packets must fit within the TCP options field, which is limited to 40 bytes, some of which are required to negotiate other TCP functionality. This requires HELLO and PKCONF to be small. HELLO requests encryption; PKCONF acknowledges the use of encryption and states the list of public key ciphers that can be used for the subsequent key exchange. Receipt of a SYN/ACK without PKCONF causes fallback to vanilla TCP.

The INIT1 message cannot be small, as it must contain the client's public key. The public key cannot fit into an option, so instead we re-purpose the data portion of one packet in each direction to carry it. The data payload is only co-opted in this way after tcpcrypt negotiation has succeeded, which ensures that key data never accidentally gets passed to applications by legacy TCP stacks. INIT2 is sent in response to INIT1 in the same way.

We use a single TCP "CRYPT" option; HELLO, PKCONF, INIT1, and INIT2 are suboptions of CRYPT. This reduces the use of scarce TCP option numbers, but more importantly it ensures that if a middlebox is going to remove one option, it should remove them all. If either host receives a TCP segment without a CRYPT option during session establishment, tcpcrypt falls back to vanilla TCP. This ensures interoperability with non-tcpcrypt-aware stacks and middleboxes that strip out unknown options. Applications can test whether tcpcrypt is used by calling *getsockopt* to request the *session ID*, which returns an error on downgraded connections.

Tcpcrypt also incorporates a re-keying mechanism, allowing session keys to evolve later in the connection to avoid using a single set of session keys for too long.

3.2 Session Caching

Applications such as the Web often establish more than one TCP connection between the same pair of hosts in rapid succession. When they do this, the amount of data

[3] One of us sometimes regrets writing the Normalizer paper.

transferred per connection can be quite small—often a few KBytes. If we have to pay the full cost of running the public key operations to establish these short-lived sessions, tcpcrypt can become a bottleneck. Fortunately we can use the same solution as SSL—cache the cryptographic state from one TCP connection and use it to bootstrap subsequent connections.

To do this we use two more CRYPT suboptions, NEXTK1 and NEXTK2, also shown in Figure 1. We cannot depend on the IP address in the SYN packet to locate the correct state because the client may have moved, or a different client may have acquired the DHCP lease used by a previous client. Thus NEXTK1 contains nine bytes of the next session ID, sid$[i + 1]$. This allows the server to verify that it has the correct cached state before using it to enable encryption. It also makes it hard for DoS attackers to flush the server's cache by spoofing packets. In the event of a cache miss, the server returns PKCONF and the protocol falls back to ordinary key exchange.

3.3 Protocol and Data Integrity

Unlike SSL, one of tcpcrypt's goals is to provide integrity protection for the TCP session itself, defending against attacks that might reset the connection [5], insert data into it, or otherwise interfere with its progress [14]. To do this, tcpcrypt adds a MAC option to every TCP packet after the INIT1/INIT2 exchange. Packets received with an incorrect or missing MAC are silently dropped.

This MAC option authenticates a segment's payload as well as a pseudo-header comprising most of the TCP header fields and options, as shown in Figure 2. We need to be pragmatic about which fields are covered by the pseudo-header. The TCP ports cannot be covered, as NATs re-write them. The MAC option is zeroed out in the pseudo-header, since it cannot authenticate itself.

Replay attacks could present a potential issue when TCP's sequence space wraps. Instead of sequence and acknowledgment numbers, the pseudo-header contains implicitly extended 64-bit values that cannot wrap. The acknowledgment number is fed separately into the MAC value, with a technique from [15], so as to improve the efficiency of retransmissions (which often acknowledge a different packet from the original).

Extended sequence numbers also solve the problem that PAWS [13] was intended to solve, so an encrypted TCP session might omit the timestamp option. This frees up eight bytes of option space; if we use a 64-bit MAC then tcpcrypt will use no more option space than most modern TCP implementations. This is particularly relevant for high performance, because when TCP's window is large it benefits from the robustness provided by Selective Acknowledgments (SACK) [19], and we do not wish to reduce their effectiveness.

<table>
<tr><td colspan="3">src port</td><td colspan="2">dst port</td><td></td></tr>
<tr><td colspan="5">seq no.</td><td>(64-bit seq)</td></tr>
<tr><td colspan="5">ack no.</td><td>(64-bit ack)</td></tr>
<tr><td>d.off.</td><td>flags</td><td>window</td><td>checksum</td><td>urg. ptr.</td><td></td></tr>
<tr><td colspan="3">options (e.g., SACK)</td><td colspan="2">MAC option</td><td></td></tr>
<tr><td colspan="5">data (encrypted)</td><td>IP length</td></tr>
</table>

Figure 2: A data packet using tcpcrypt. Dashed quantities are not transmitted by TCP though included in the MAC, along with shaded fields.

More subtly, we need to be careful about middleboxes that modify packets. If an implementation does send the timestamp option, tcpcrypt will normalize it to zero in the pseudoheader, as OpenBSD's pf [10] modulates its value. All the other options that are commonly modified occur only in the SYN or SYN-ACK, so do not present a problem. Tcpcrypt does provide a secure timestamp-like suboption to CRYPT called SYNC. SYNC is covered by the MAC, but fuzzes the clock to avoid the reasons for which pf needs to modulate the timestamp's value. Moreover, the SYNC option is only required for keepalive packets and during re-keying when the connection is otherwise idle. In both cases there is no need for SACK blocks, so the option space is less precious.

Packets with the TCP RST bit set present the final challenge. For full protection, after session establishment we would prefer to drop RST packets that do not contain a valid MAC option. However, RST is TCP's mechanism for informing one side of a connection that the other side no longer has any state for the connection. Under such circumstances it is impossible for a legitimate host to generate a RST packet with the MAC option. Tcpcrypt's default behavior is to reset the connection when receiving a RST with no MAC, so long as it passes the OS's sequence number validity checks. However, some applications (notably BGP routing) have a much stronger requirement to protect against connection resets. For these applications we support a setsockopt that mandates RST packets carry a valid MAC. Such connections will take a long time to time out if one side loses state; however, applications such as BGP and SSH that might require such protection also typically use application-level keepalives to detect liveness and so tear down stale connections.

3.4 Application Awareness

Tcpcrypt serves a dual role: for legacy applications it protects against passive eavesdroppers; for tcpcrypt-aware applications it enables stronger protection, as we will discuss below. However, it is important to avoid a duplication of functionality.

Consider a tcpcrypt-aware web browser on a tcpcrypt-

capable host that wishes to make an authenticated connection to a web server. The browser might prefer tcpcrypt because of the availability of better password authentication methods, but only if the web server also supports it. Otherwise, it wishes to fall back to SSL.

A potential problem occurs when the client connects to a legacy web server process running on a tcpcrypt-capable host. Under such circumstances we do not wish to use both unauthenticated tcpcrypt and authenticated SSL encryption, which would be the default behavior. Rather, the web browser wishes the tcpcrypt negotiation in the SYN exchange to fail unless *both* the host and the web server process can use the tcpcrypt-based authentication.

To get this correct fallback behavior, the HELLO option includes a "*Mandatory Application-Aware*" bit. When set, this bit indicates to the server that it *must not* enable tcpcrypt encryption unless the server application has informed the stack that it is tcpcrypt-aware. The process uses a `setsockopt` on the listening socket to do this. Our enhanced SSL implementation that uses this mechanism is described in Section 5.3.

Tcpcrypt also includes a second "*Advisory Application-Aware*" bit in both the HELLO and PKCONF options. This is used for each side to indicate to the other that the application is tcpcrypt-aware. This is used when applications want to perform authentication over tcpcrypt if the other side is also tcpcrypt-aware, but where it is not necessary to fall back to an unencrypted session if the other side is not tcpcrypt-aware. For example, many websites with low security requirements use HTTP Digest authentication. Such websites can still use HTTP Digest authentication over tcpcrypt (though we would not advise it), but if both the client and server applications are tcpcrypt-aware, it would be possible to drop in CMAC-based mutual authentication instead. However, the client needs to know that the server can do this before sending the HTTP request, and the "*Advisory Application-Aware*" bit provides this information. It is set via a `setsockopt` before calling `connect` and retrieved at the other side via `getsockopt` after the connection handshake completes.

4 Authentication examples

User authentication is an area in which there exist simple and well-known techniques qualitatively superior to those in widespread use. For instance, websites typically request passwords be sent straight to the server. As a result, we see many successful phishing attacks. Almost all of these attacks could very easily be defeated with known techniques, were it not for issues of backwards compatibility in protocols and user interfaces. Thus, there are strong incentives to make improvements to authentication in the web and other applications.

To realize this shift to better authentication protocols we need innovation in user-interface design. Currently, HTTP digest authentication, while better than plaintext passwords, is seldom used because web developers shun browsers' ugly gray popup boxes. The challenge is to allow some aesthetic control by web sites while simultaneously ensuring password entry is unambiguously differentiated from web forms (or anything else accessible by JavaScript). Tcpcrypt itself obviously cannot improve user interfaces; the aim is to ensure that when improvements do happen, they can easily be integrated with tcpcrypt to provide security against active attackers.

The hook tcpcrypt provides to application-level authentication is the session ID. This section gives a few examples of how session IDs can be used, assuming the ability to display certificate names and to input passwords from a user securely. Though these examples require modifications to applications, such enhancements can be deployed incrementally using tcpcrypt's Application-Aware bits described in the previous section.

Note that the prevalence of weak authentication makes for some very low-hanging fruit. We do not claim these obvious and well-known fixes as contributions. Nor do we mean to imply that these techniques would not work with application-layer traffic encryption were we to enhance SSL. Our point is merely to illustrate the generality of the session ID abstraction and to help substantiate our claim that tcpcrypt provides encryption as a general building block suitable for a wide range of applications.

The key properties we rely on are that 1) if both ends of a connection see the same session ID, then the session data's confidentiality and integrity are ensured, and 2) session IDs are unique over all time with overwhelming probability, even when one end of a connection is malicious.

4.1 Certificate-based authentication

One common basis for server authentication is certificates, such as the X.509 certificates employed by SSL. (This model may become even more prevalent if DNSSEC gains widespread deployment.) In this model, each server S has a long-lived public key, K_S, certified by a trusted authority to belong to a particular common name and organization. The common name or organization can then be presented to the user to inform her of whom she is communicating with.

Certificates permit a trivial authentication protocol:

$$S \to C: \; K_S, \text{Certificate}, \textsc{Sign}\left(K_S^{-1}, \text{Session ID}\right)$$

The server simply signs the session ID, thereby proving it owns one end of the connection, ensuring confidentiality

of messages sent by the client and integrity of those sent by the server.

The problem with the above protocol is the cost of the SIGN function, which can be comparable to public-key decryption. The cost for the server to compute such a signature for every new client would be comparable to setting up an SSL connection, which is one of the factors dissuading people from using SSL ubiquitously today. While there do exist some faster signature schemes (*e.g.,* [7]), the certificate authorities may not be willing to endorse non-standard algorithms.

Fortunately, there is a better approach. Heavily loaded servers can amortize the cost of a single signature over many sessions by signing a batch of session IDs. Session IDs are not secret, so disclosing a batch of them to each client is not a problem.

Once a single session has been authenticated, the same pair of machines can use the existing connection to bootstrap authentication of other sessions using only symmetric cryptography. For instance, they can exchange a MAC key and use it to authenticate future session IDs.

4.2 Weak password authentication

Often two connection endpoints share a secret. For instance, a user may remember a password, and a server may store some secret derived from the password. Today, all too often passwords simply authenticate the user to the server and not vice versa. As a basic principle, if we deploy new authentication mechanisms, *any time a user types a password, it should mutually authenticate the client and server to each other*. There is simply no reason ever to use a password to authenticate only one endpoint of a communication. Even if the other end is a server with an X.509 certificate, the certificate may have been fraudulently obtained, or it may be for a "typo" domain name similar enough to the desired one that the user doesn't notice the error.

When a server, S, is under severe performance constraints, it can perform password authentication using symmetric cryptography. For instance, S may store the secret hash value of a user's password, $h = H(\text{salt}, \text{realm}, \text{password})$; a client C can query S for the non-secret salt, then compute h from a user-supplied password. Section 6 benchmarks the following trivial authentication protocol for such settings:

$$C \rightarrow S: \ \text{MAC}\,(h, \texttt{TAG_CLIENT} \| \text{Session ID})$$
$$S \rightarrow C: \ \text{MAC}\,(h, \texttt{TAG_SERVER} \| \text{Session ID})$$

This protocol is no more costly or hard to implement than digest authentication [6] (in fact, possibly easier, as it requires no randomness beyond that already reflected in the Session ID). Yet it provides better guarantees, namely *mutual* authentication of S to C as well as integrity and confidentiality of all session data. The protocol assures both C and S that the other end of the connection knows h. Such a guarantee is different from and complements that provided by certificates—*i.e.,* that a server owns a particular domain name. Domain-name certificates offer important protection in many contexts, but this session-ID-based protocol offers protection even when users do not remember the correct domain name.

We note that even if an attacker hijacks DNS to impersonate S, our protocol is resistant to phishing for users with good passwords. The protocol can be viewed as endorsing the session ID with h; since session IDs are unique over time, the attacker may obtain $\text{MAC}(h, \texttt{TAG_CLIENT} \| \text{Session ID})$, but this value is meaningless in the context of any other connection.

Unfortunately, while the above protocol would be categorically superior to plaintext passwords and digest authentication, we still do not advocate using it except for servers on which stronger authentication would require too much CPU time. The problem is that an attacker who impersonates the server to obtain the first message can then mount an offline dictionary attack on the password, leveraging the single message exchange to guess arbitrarily many passwords. Such an attack may be detectable if the attacker cannot crack the password in time to mount a transparent man-in-the-middle attack—but people are used to clicking reload sometimes when web sites fail and will not be concerned by a single connection failure.

4.3 Strong password authentication

Fortunately, as detailed in Section 6, any site that can afford to use SSL today can afford to use a strong password authentication scheme with tcpcrypt. Here we give a simple example of a Password-Authenticated Key-Exchange (PAKE) protocol that that, while considerably more expensive than the previous weak protocol, can nonetheless be implemented with far less overhead than SSL imposes today.

We use a protocol termed $\texttt{PAKE}_2^+$ in [4]. The protocol relies on several system-wide parameter choices: a group $\mathbf{G}$ of prime order q (on which the computational Diffie Hellman problem is hard); a generator g of $\mathbf{G}$; two randomly-chosen elements of $\mathbf{G}$, U and V; two cryptographic hash functions, H_0 and H_1, mapping strings to elements of $\mathbf{Z}_q$; and finally, another hash function, H, onto bit strings the size of a MAC key. At the time a user registers for an account, her client computes:

$$\pi_0 = H_0(\text{password, user name, server name})$$
$$\pi_1 = H_1(\text{password, user name, server name})$$
$$L = g^{\pi_1}$$

The server stores π_0 and L, but never sees π_1. To authenticate a session, the client chooses a random element $\alpha \in \mathbf{Z}_q$ and the server chooses a random element $\beta \in \mathbf{Z}_q$. The two then engage in the following protocol:

$C \rightarrow S: \; g^{\alpha} U^{\pi_0}$
$S \rightarrow C: \; g^{\beta} V^{\pi_0}$

At this point, both sides compute $g^{\alpha\beta}$. They can do this by computing either $U^{-\pi_0}$ or $V^{-\pi_0}$ and using it to revert to a regular Diffie-Hellman key exchange. Then both sides compute $g^{\pi_1\beta}$. The client can do this because it knows g^{β} and π_1. The server can do this because it knows: $L = g^{\pi_1}$ and β. Finally, both sides compute:

$$h = H\left(\pi_0, g^{\alpha}, b^{\beta}, g^{\alpha\beta}, g^{\pi_1\beta}\right)$$

Using h they complete the password authentication protocol of the previous section, but now the order of messages doesn't matter (the client and server can each transmit one of these messages before receiving the other to reduce latency):

$S \rightarrow C: \; \text{MAC}\left(h, \texttt{TAG_SERVER} \| \text{Session ID}\right)$
$C \rightarrow S: \; \text{MAC}\left(h, \texttt{TAG_CLIENT} \| \text{Session ID}\right)$

While this protocol is considerably more expensive than the one in the previous section, it has the benefit of protecting users with weak passwords; each guess at the password requires a separate network interaction with a party that knows either the password or π_0 and L. Moreover, the protocol is still cheaper than SSL (even combined with tcpcrypt key negotiation). Therefore, we believe it is suitable for use in any application that uses both passwords and SSL.

It is an open question whether we can design password authentication protocols that are highly efficient at the server and offload most of the work to the client. However, should we devise such protocols, they can be deployed after the fact, without modification to tcpcrypt itself. The session ID abstraction nicely separates tcpcrypt's confidentiality and integrity properties, which are solved problems, from authentication, where further innovation may be needed.

5 Implementation

To validate the protocol design and verify its performance, we implemented tcpcrypt in the Linux kernel. We also implemented tcpcrypt as a user-space daemon using divert sockets; this allows tcpcrypt to be deployed easily without requiring any kernel changes. Finally we implemented a range of application authentication mechanisms over tcpcrypt.

5.1 Linux kernel implementation

Our kernel implementation of tcpcrypt consists of a 4,000-line loadable module and 70 lines added to the core Linux 2.6.32 kernel to add the necessary hooks. For RSA support, we ported OpenSSL v0.9.8l to the Linux kernel. This required about 400 lines of glue code to export RSA as a Linux crypto module. We also exposed OpenSSL's SHA1 as we found it to perform twice as fast as Linux's implementation.

During the implementation, it became clear that tcpcrypt is incompatible with TCP segmentation offloading, as supported in some modern NICs. As tcpcrypt has to copy the packet to memory to encrypt the data and compute the MAC, segmenting it during this process does not add significant overhead. However, a server running so close to its performance limits that it requires segmentation offloading would likely want to disable tcpcrypt.

5.2 Portable userspace implementation

Our userspace tcpcrypt implementation uses divert sockets to access TCP packets entering and leaving the host. Firewall rules select the packets to be diverted, leaving the kernel unchanged. FreeBSD's NAT (`natd`) is implemented this way. The main advantages of this approach are portability and ease of deployment. Our code is 7,000 lines. We have tested it on MacOS X, FreeBSD and Linux.

The userspace implementation is obviously slower than the native kernel implementation, but it is ideal for early deployment without support from OS vendors. If tcpcrypt is successful and ships in major operating systems, it will still be a long time before older hosts are upgraded. The userspace implementation provides a good interim solution. It can also be run on middleboxes such as firewalls or home gateways to protect traffic to and from legacy local hosts against passive eavesdropping.

The userspace implementation is more complicated than the kernel one as it must track connections, duplicate much of TCP's state machine, calculate checksums again, and rewrite sequence and acknowledgment numbers since we use some bytes of the payload for INIT messages. In SYNs the MSS is reduced to allow space to add the MAC to subsequent packets. In addition, the sending of application data must be delayed until the tcpcrypt handshake completes, which we do by modulating the receive window. Finally, we implement IPC calls to provide the equivalent of *getsockopt*, so the application can extract the session ID to perform authentication.

5.3 Integrating tcpcrypt and OpenSSL

If tcpcrypt were enabled by default, then an SSL connection between two tcpcrypt hosts would duplicate effort doing both tcpcrypt and SSL key exchange and encryption. Tcpcrypt's *Mandatory Application-Aware* bit avoids this duplication. To verify this mechanism and to compare the full performance of Apache running SSL-over-tcpcrypt using batch-signing to that of vanilla SSL, we implemented tcpcrypt support within the OpenSSL

v0.9.81e library. We did not modify OpenSSL's API or require applications to set specific parameters to gain the benefits of tcpcrypt and batch-signing—our library is a drop-in replacement for OpenSSL.

Our implementation uses the tcpcrypt `setsockopt` to notify the kernel that the application supports tcpcrypt, setting the *Mandatory Application-Aware* bit during the handshake. After the TCP handshake, either the session is encrypted and both sides support tcpcrypt-based authentication, or the connection has fallen back to vanilla TCP. The library code then queries with `getsockopt` to get the session ID. If this returns an error, it falls back to SSL's handshake, otherwise it batch-signs the session ID and sends it to the client.

We modified OpenSSL's BIO layer to call the necessary `setsockopt` for setting the application bit. The SSL layer, *i.e.,* `SSL_accept` and `SSL_connect`, then deals with the signatures. Thus, so long as the application uses the BIO API, no change to the application is needed to use tcpcrypt-based authentication instead of SSL authentication.

Things are not quite so clean if application programmers manually create sockets using the BSD socket APIs instead of BIO, feeding them directly into `SSL_accept` and `SSL_connect`. These sockets will not have the necessary options set, and so tcpcrypt would disable itself even though the SSL library is capable. In such cases, if upgrading the application is not possible, then a `sysctl` could be used to set the application bit on by default on specific TCP ports.

Batch signing is implemented per SSL context. A single worker thread (per SSL context) waits on a semaphore for work and batch signs all session IDs it finds on its work queue. The signer thread then wakes up all threads corresponding to the session IDs signed. For batch signing to work, the SSL server must be multithreaded. We note that this implementation naturally scales depending on load: if a single client needs a signature, it is produced right away; when under load, multiple client session IDs will be batch signed to amortize cost. Our OpenSSL patch and batch signing code total 700 lines of code.

5.4 Password based authentication

We implemented the weak password authentication scheme in Section 4.2 as well as the strong scheme from Section 4.3. The weak scheme uses CMAC-AES as the MAC, and employs IBM's CMAC patch [21] for OpenSSL. We implemented the strong authentication scheme ourselves (500 lines of code) using OpenSSL's built-in support for NIST Prime-Curve P-256.

6 Performance and compatibility

If we are to achieve our ultimate goal of encrypting almost all Internet traffic, then the cost of doing so must be sufficiently low that the cost/benefit trade-off makes sense, even when the benefits are small. What then are the costs of running tcpcrypt? Roughly, the performance cost breaks down as follows:

- The cost of the tcpcrypt key exchange.
- The cost of encrypting and MACing every packet on the wire.
- The cost of authentication over tcpcrypt, for applications that choose to authenticate.

We must demonstrate that the first two are small enough they will not significantly degrade the performance of the vast majority of servers (clients are rarely the bottleneck, as they handle only a few connections per second at most). We must also demonstrate that the third is at least as cheap as current deployed solutions.

In addition, we must also demonstrate compatibility. Tcpcrypt must not cause connections to fail that would succeed without tcpcrypt.

6.1 Connection setup rate

Just how fast do servers need to accept connections in practice? It is hard to get firm numbers. *YouTube* gets 1 billion hits per day [12], thus averaging about 11,500 hits per second. *Facebook* currently gets about 260 billion page views per month [20], or around 100,000 per second. Of course a page may require more than one TCP connection, but with HTTP/1.1 the number will be fairly small. Facebook also has over 30,000 servers [24]. Not all these are front-end servers, but even so it becomes clear that the number of connections that need to be handled per second on each server is unlikely to be more than a few thousand.

To get another perspective, we can examine what an untuned operating system running an untuned web server can achieve. This tells us how default configurations perform, and so what a typical server administrator might expect. Our test machines are eight-core (two Intel Xeon X5355 CPUs) running Linux 2.6.32. Each has 13 1Gb/s NICs connected to client hosts via a LAN. Multiple clients and parallel connections are needed to saturate the server. Untuned, these servers can handle 35,500 TCP connections per second in a simple connection setup and teardown test, or 28,400 connections per second running Apache serving a small static file.

To determine the effect of tcpcrypt, first we need a better control experiment because the untuned numbers above, although typical of most real-world installations,

fail to fully utilize the machine, leaving some idle time. It took considerable tuning[4] to get the connection setup and teardown test to saturate all the cores. Such a setup is not realistic for normal operation, but we wish to compare against the best-case vanilla TCP, not one that leaves unused CPU cycles. We will compare this optimized TCP against SSL and tcpcrypt.

We expect tcpcrypt to slow down TCP's connection throughput in two main ways. First, uncached tcpcrypt connections use public key operations to setup a connection. This cost is predominantly born by clients, which perform the more expensive RSA decryption operation. We use 2048-bit RSA-3 keys in all benchmarks.

Second, packets are MACed and thus require more CPU cycles and memory accesses. Even with connection caching, which avoids the need for public key cipher operations, four out of six of the packets in an `accept`/`close` cycle are MACed (two ACKs and two FINs). We therefore expect a performance degradation both in the uncached and cached connection cases, though uncached connections will be more expensive.

We expect SSL to perform less well than tcpcrypt for two reasons. First, it requires more RTTs to complete a connection because SSL's handshake can only start after TCP's handshake. More notably, uncached SSL connections should be much slower than tcpcrypt's because an SSL server performs the more expensive RSA decryption operation. However SSL also authenticates the server, so this is not an apples-to-apples comparison. We shall examine the cost of tcpcrypt's authentication in Section 6.2.

Protocol	Connection rate (conn/s) Native	Divert
TCP server	98,434	61,515
tcpcrypt server (cached)	70,044	38,832
tcpcrypt server (uncached)	27,070	21,908
SSL server (cached)	39,785	27,348
SSL server (uncached)	754	743
tcpcrypt client (uncached)	794	749

Table 1: Connection setup rate of tcpcrypt.

Table 1 shows the results. Both the cached case (same client reconnecting) and uncached case (new client, requiring public key cipher operations) are shown. The two columns benchmark our two tcpcrypt implementations: the kernel one ("Native") and the userspace divert socket one. To get divert numbers for TCP and SSL, we divert all traffic to userspace and back to the kernel; although this isn't useful, it allows us to separate out the different costs and see the overhead of the divert socket separately from additional protocol mechanism in the tcpcrypt userspace implementation.

Tcpcrypt outperforms SSL in the uncached case by a large margin due to reversing the asymmetric RSA costs; the client bears this cost. Tcpcrypt's cached performance is also better than SSL. We note that our kernel implementation is not fully optimized, so it may well be possible to get even greater performance. For example, we could encrypt and MAC data while copying it from userspace rather than doing it on a later pass. This would be an optimization similar to that for checksum calculation already used in Linux.

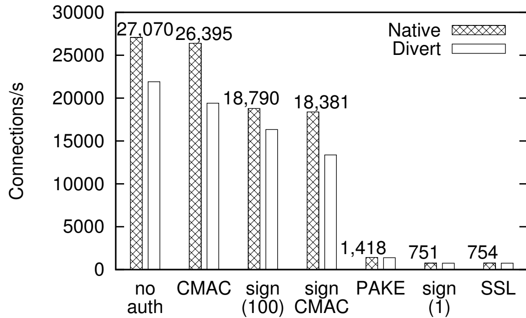
While TCP can be up to 41% faster for cached connections and 3.6× faster for uncached ones, we believe that the absolute performance numbers of tcpcrypt have their own merit. Recall that a heavily loaded website like Youtube averages 11,500 connections per second and tcpcrypt should be able to sustain such high load. Also recall that our untuned default configuration server can only handle 35,500 connections running the same benchmark of Table 1, also a target that tcpcrypt can meet if some sessions are cached.

The divert socket implementation is slower than our kernel one due to the multiple copies needed for each packet, from the kernel to userspace, and then back to the kernel. Furthermore the userspace implementation needs to (wastefully) duplicate TCP functionality already present in the kernel such as checksum calculations and protocol control block lookups. However, we believe that the absolute performance numbers of our divert socket implementation are sufficient for many situations, especially on clients, where simple installation may be a priority over performance.

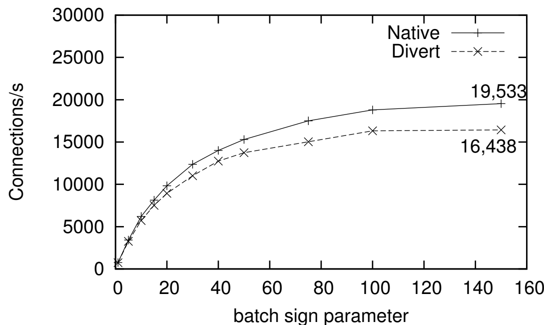
6.2 Authenticated connection setup rate

While Table 1 included SSL as a reference point, it cannot be used to directly compare the two systems because SSL performs authentication by default and thus is stronger than unauthenticated tcpcrypt. As tcpcrypt leaves authentication to applications, we are free to examine different authentication schemes. Our authentication benchmarks cover: tcpcrypt with batch signing (SSL replacement), CMAC password-based mutual authentication (vulnerable to offline dictionary attacks), batch signing combined with CMAC, and PAKE_2^+ password-based mutual authentication (both resistant to offline dictionary attacks). The benchmark and setup is identical to our previous benchmark, but with added application-level authentication after connection setup. We expect tcpcrypt with batch signing to outperform SSL when

[4] This involved running multiple instances of the benchmark on different ports to avoid kernel locks on `accept`. We set the affinity of each benchmark to one CPU, and used a different NIC per benchmark, with the NIC's interrupt affinity set to the same CPU as the benchmark using the NIC. This resolved in optimal packet scheduling and load balancing that finally brought the system to zero idle time.



(a) Authentication performance comparison.



(b) Batch signing.

Figure 3: tcpcrypt's authenticated connection setup rate.

batching more than one request, as RSA signatures will be amortized. We expect CMAC to outperform RSA-based authentication, because it uses symmetric cryptography only. Our PAKE implementation is so far unoptimized, but even so we expect it to be faster than RSA because it replaces the expensive RSA signature with a few elliptic-curve operations.

Figure 3 shows tcpcrypt's authenticated connection setup rate when using our kernel implementation ("Native") and our userspace divert socket one. Batch signing performs differently depending on the size of the batch and Figure 3(b) shows how this scales. Most of the benefits of batch signing arise even with a parameter as low as 100, a number of concurrent clients easily reached when the server is under load. Figure 3(a) clearly shows that there is a range of performance characteristics which applications may choose from. With SSL instead, applications are forced to use relatively low performance one-way authentication. Clearly, one size does not fit all. With tcpcrypt, applications can choose any combination of one-way or two-way authentication and higher performance at lower security or lower performance at higher security. For example, a busy web forum might choose CMAC for its authentication as it requires two-way authentication and high performance, but perhaps is not so security-critical that it needs to thwart offline dictionary attacks. This setup would perform 36x faster than SSL on uncached connections, providing stronger (two-way) authentication. A bank instead, might choose PAKE for its authentication, performing slower, but still twice as fast as SSL. Alternatively if a certificate is available, signing plus CMAC could be 24x faster than SSL and still resist offline dictionary attacks. A site requiring only one-way authentication, like a checkout from an online shop that does not require login, can perform up to 26x faster than SSL when loaded and handling over 150 concurrent requests. Tcpcrypt with batch signing is therefore a viable drop-in replacement for SSL, as in all cases its connection setup performance is superior (we shall examine data throughput in Section 6.4). Authentication adds little cost to tcpcrypt: 2% penalty with CMAC or 28% with batch signing under load. We believe this performance to be practical for many servers.

For most clients the performance of the divert socket implementation will be sufficient, providing an easily installed alternative.

Hardware is often used to offload expensive public key cryptography. For example, Sun's UltraSPARC T1 has a Modular Arithmetic Unit for RSA, and can do 2,300 2048-bit signatures per second using all 32 cores [18]. Tcpcrypt outperforms this using only eight general purpose cores, showing how careful protocol design can avoid the need to throw hardware (and money) at the problem. We argue that offloading asymmetric encryption is no longer needed for network encryption.

Protocol	Connect time (ms) LAN	WAN
TCP	0.2	105
Tcpcrypt cached	0.3	103
Tcpcrypt not cached	11.3	219
Tcpcrypt CMAC	11.4	320
Tcpcrypt PAKE	15.2	426
SSL cached	0.7	210
SSL not cached	11.6	321

Table 2: Connection setup time.

6.3 Connection latency

Throughput is not the only important metric—connection setup latency is also important. We compare the connection setup time from the client's point of view for TCP, SSL and tcpcrypt. We expect tcpcrypt to setup connections faster than SSL because tcpcrypt's handshake requires fewer round trips. Table 2 shows the time to establish a connection on a LAN (0.2ms RTT) and on a WAN (100ms RTT).

When the connection is cached, tcpcrypt adds very

little delay to TCP because no extra RTTs are needed. Tcpcrypt does extra work to advance keys and MAC the ACK, hence it takes fractionally longer. SSL cached takes considerably longer because its negotiation can only start after TCP's handshake finishes whereas tcpcrypt uses the three-way handshake. In the non-cached case tcpcrypt and SSL perform similarly on the LAN as RSA dominates the cost. The main difference is that tcpcrypt is client-limited whereas SSL is server-limited. On the WAN, RTT dominates; tcpcrypt costs one RTT more than TCP, but one RTT less than SSL as it needs fewer messages to complete the handshake. Authenticating an uncached tcpcrypt connection, for example using CMAC or PAKE, adds extra latency.

With batch-signing there might be a concern that the queuing of requests to be signed might add extra latency. In fact this is not the case—our implementation signs whatever queue is available as fast as it can. Even the fact that tcpcrypt with signing requires two RSA operations does not add to latency—the expensive decrypt operation on the client takes place in parallel with the sign operation on the server, so negligible extra latency is required beyond the extra RTT needed for authentication.

The main effect of batch signing is in fact to reduce latency as the server becomes loaded. This is shown in Figure 4, which graphs connection latency against the number of connections per second the server handles. As the load increases eventually the server saturates and the latency increases extremely rapidly. The figure shows SSL latency and tcpcrypt latency when the maximum batch size has been artificially limited to 1, 5 and 10. SSL and tcpcrypt with a batch size of one are indistinguishable on this graph, so we only plot one line. It is clear that when the server has CPU cycles to spare, the batch size has no adverse effect on latency. In fact, quite the reverse—batching reduces the variance (the plot shows 10th and 90th percentiles as error bars), because short-term variations in arrival rate map into variation in batch size rather than variation in CPU load. More importantly, allowing larger batch sizes allows the server to saturate much later, and so maintain this low latency across a much wider range of server workloads.

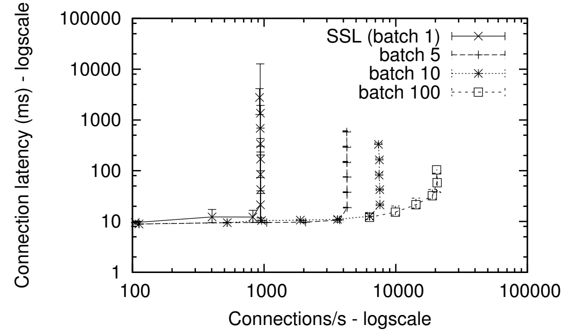


Figure 4: Latency as connection rate increases.

Protocol	Transfer Throughput (Mbit/s) Native	Divert
TCP	12,954+	3,357
tcpcrypt AES-SHA1	3,968	1,752
SSL AES-SHA1	3,692	1,939

Table 3: tcpcrypt's data throughput.

6.4 Data transfer rates

We now account for the cost of symmetric encryption and determine the maximum data throughput one can expect with tcpcrypt. We benchmark data throughput when transmitted with TCP, tcpcrypt and SSL. To fully saturate the CPU we ran one benchmark program per core and NIC pair, setting the affinity of the benchmark and NIC to a particular core. Otherwise, packet scheduling was suboptimal resulting in idle time. We expect SSL and tcpcrypt performance to be similar as both are doing AES128 and HMAC-SHA1. Obviously, vanilla TCP will be fastest as it need not encrypt or MAC.

Table 3 shows the data throughput of tcpcrypt, for our kernel implementation ("Native") and our userspace divert socket one. We were unable to saturate the CPU on the TCP benchmark (11% idle time) as we saturated all available NICs on the server. Tcpcrypt outperforms SSL by 7.4%. This was unexpected as the two essentially perform the same tasks: AES and SHA1. We are using different implementations for AES (Linux's kernel *vs.* OpenSSL) though we found the two to perform similarly when benchmarked individually. The fundamental differences between tcpcrypt and SSL are that SSL must do its own data segmentation and encapsulation (in addition to TCP's) thus needs more work than tcpcrypt. SSL MACs at a message boundary which can span multiple packets, whereas tcpcrypt must MAC once per packet. Tcpcrypt is MACing slightly more data as it includes packet headers, though the cost of SSL's message encapsulation seems to outweigh the additional bytes MACed by tcpcrypt. Overall, however, CPUs are powerful enough to fully encrypt a one Gigabit link, and in fact even more. Client machines seldom have more capacity than that, and even our userspace implementation provides sufficient performance for those cases.

Most relevant to servers, higher rates are possible by using faster ciphers and MACs; tcpcrypt achieves 7,486Mbit/s using Salsa20/12 and UMAC. High-speed AES is possible too now that AES-enhanced CPUs are becoming ubiquitous, like Intel's Westmere CPU [8], Sun's UltraSPARC T2 [2] and VIA's processors [1]. On a dual-core 3.33GHz desktop i5 with a 10Gb/s NIC,

Protocol	Apache, static content (req/s) Native	Divert
TCP	60,156	27,196
tcpcrypt (cached)	42,440	20,034
tcpcrypt (uncached)	19,153	14,215
SSL (cached)	19,787	12,063
SSL (uncached)	737	705

Table 4: Apache performance serving static content.

tcpcrypt performed 8,835Mbit/s using AES-UMAC, even without TCP segmentation offloading and optimizations in tcpcrypt. As an experiment, we were able to saturate 10Gb/s by using jumbo frames or by overclocking the box to 3.75GHz. We thus soon expect CPUs that will permit 10Gig AES networking—in fact, this is likely possible today if a six-core server i5 is used.

6.5 Application performance: Apache

We now study the overhead of tcpcrypt when used in a real application. We test the Apache web server (v2.2.11) serving a 44 byte static file. This setup has low application overhead, emphasizing overhead imposed by the networking stack. With a default configuration, our server can handle 28,400 requests per second though the CPUs remain unsaturated. To fully saturate CPUs, we must run multiple Apache instances, each on a different TCP port, serving traffic on a different NIC. Based on our microbenchmarks, we expect tcpcrypt to outperform SSL and have lower performance than TCP. We do not perform any authentication on this tcpcrypt benchmark, so SSL provides stronger guarantees in this case. However, as discussed earlier, authentication can be added to tcpcrypt at a relatively low cost if needed.

Table 4 shows the results of our Apache benchmark. Because real-world web traffic is a mix of new and returning clients, connection setup can quickly become a bottleneck for SSL. Tcpcrypt, on the other hand, maintains a high connection rate (31% of native TCP) even for new clients. Note also that the case of small, static files is a worst-case benchmark for connection setup. We tried benchmarking WordPress, a more CPU-intensive application. Neither tcpcrypt nor SSL caused a measurable slowdown. This test demonstrates that ubiquitous encryption is feasible when the application is the bottleneck, and in most cases even if it is not.

6.6 Compatibility

Incremental deployment is one of our chief goals. Essentially this entails gracefully falling back to TCP so that connections are guaranteed to succeed. Users will not enable tcpcrypt if doing causes their connections to fail. Tcpcrypt falls back gracefully so long as packets with the CRYPT option do not get dropped. Otherwise, tcpcrypt might indefinitely send SYN packets with the CRYPT option, and the connection would fail when it would succeed using a virgin SYN packet. To gauge whether this is a problem, we initiated tcpcrypt connections to the top 10,000 sites listed on Alexa. Specifically, we sent a SYN with the CRYPT-HELLO option set, expecting to get a SYN-ACK back. If not, we considered the packet dropped. We retransmitted SYNs to detect packet loss. This gives a rough estimate of how many connections would fail because of tcpcrypt.

Of the Alexa top 10,000 sites, we found 15 (0.015%) that did not respond with a SYN-ACK to a tcpcrypt SYN. Of these, three were on the same network. Given such a low failure rate, we are optimistic that tcpcrypt will work most of the time and can be safely deployed. However, by default, tcpcrypt will try to revert to standard TCP in case it does not receive a SYN-ACK after sending a few tcpcrypt SYNs to ensure reachability.

We do not expect tcpcrypt to suffer ECN's fate in terms of compatibility. ECN used reserved bits in the TCP header which would trigger IDSs and cause undefined behavior. Instead, tcpcrypt uses options as dictated by TCP's specification and is not anomalous in any way—for instance, even during re-keying the protocol design ensures that retransmissions always produce the same payload bytes for a given range of sequence numbers. We thus believe that tcpcrypt can safely be deployed on today's Internet as it will, for the majority of users, provide stronger security without breaking connections or noticeably reducing performance.

7 Related work

We categorize related work based on the networking stack layer it operates in. The network layer is dominated by IPSec-based solutions. IPSec [16] encrypts all data above the network layer. However, IPSec has not enjoyed widespread deployment and use, so a reasonable fear is that tcpcrypt could endure the same fate. Fortunately, several factors make it easier to deploy tcpcrypt and provide greater incentive to do so, leaving us some hope that ubiquitous encryption can succeed at the transport layer even if it has not at the network layer.

A big challenge to IPSec is that it breaks middleboxes that require access to the transport layer. Given the increasing prevalence of NAT in particular, this excludes a large portion of the population from using IPSec. Tcpcrypt, by contrast, operates at the transport layer and so avoids these problems. Another challenge for IPSec is that it is hard to create a notion of a "session" in a connection-less environment (the network layer). Thus, while IPSec is good at authenticating hosts to one an-

other for purposes such as virtual private networks, it would be difficult and cumbersome to authenticate individual users, processes, and connections between hosts. Moreover, some transport-level security issues, such as protecting against wrapped acknowledgment numbers, are harder to reason about in IPSec.

Conversely, there are several incentives for deploying tcpcrypt that have no analogue with IPSec. One is that it can be integrated in a backwards-compatible way with SSL and significantly increase performance. By contrast, SSL over IPSec would require double-encryption, reducing performance. Second, TCP multipath requires a means of authenticating the same endpoint with multiple IP addresses, which tcpcrypt makes much easier. That said, tcpcrypt is less general than IPSec, which encrypts everything above IP, including UDP.

Better Than Nothing Security (BTNS) [26] is IPSec without a PKI, thus providing no security guarantees against active attackers. This is similar to default tcpcrypt. However, tcpcrypt additionally exposes the necessary hooks so that applications can perform authenticate in a variety of ways to guarantee security. Opportunistic encryption using IKE [23] specifies how to use IPSec with certificates obtained from DNSSEC. Tcpcrypt would need application support to integrate with DNSSEC.

We found no privacy solutions integrated into the transport layer. There are, however, integrity solutions. TCP MD5 [11] and AO [27] provide authentication and integrity protection within TCP. Tcpcrypt provides more functionality than these options by providing encryption. Moreover, tcpcrypt is fundamentally different as it requires no user setup. The session is established using ephemeral keys, and authentication can happen over the session itself. TCP MD5 and AO require establishing pre-shared secrets through out-of-band means. The main use of TCP MD5 and AO is to protect manually configured BGP sessions, which tcpcrypt can do as well by disabling unauthenticated RST packets. Also, TCP AO does not interoperate with NATs (which is okay for its intended use, as BGP is not usually spoken through NATs).

The dominant encryption solution above the transport layer is SSL [22]. Tcpcrypt offers a number of benefits over SSL, including better server performance, intrinsic forward secrecy, and integrity protection for the TCP session itself. Tcpcrypt is also more general, as it supports arbitrary authentication mechanisms and does not require a PKI. Finally, tcpcrypt is backward compatible with legacy applications and legacy hosts, which should ease ubiquitous deployment. Being more general, tcpcrypt can be used as a drop-in replacement for SSL, and we have in fact produced an SSL library that falls back to SSL if tcpcrypt is unavailable.

ObsTCP [17] also aims to provide opportunistic encryption, but is only designed to provide security in aggregate, not for specifically targeted connections. The author states, "We continue to advocate TLS as the only user facing transport security," meaning ObsTCP will duplicate encryption done by TLS, not protect transport headers, and not integrate with application-level authentication. ObsTCP requires no new TCP options and no extra round trips for connection setup, but the downside is that applications must be modified and that the first connection between two hosts remains unencrypted unless one knows that the other supports ObsTCP.

While tcpcrypt combines only well-known techniques, no other existing protocol can accomplish all of its goals. Specifically, tcpcrypt can be incrementally deployed on today's Internet, works out-of-the-box (even through NATs) without manual configuration, provides high enough performance to be on by default, and allows applications to integrate transport-layer security with arbitrary higher-level authentication techniques. The Internet demands higher security, hardware is ready for it, and the cryptographic techniques were waiting to be pieced together; tcpcrypt does so, and we believe our evaluation shows it could be readily deployed.

8 Conclusion

Tcpcrypt demonstrates that ubiquitous encryption of TCP traffic is technically feasible on modern hardware. By leveraging the asymmetry of common public key ciphers, it is possible for a server to accept and service around 20,000 tcpcrypt connections per second without session caching. Even higher rates are possible with caching. Data transfer rates are not an issue either; AES-SHA1 encryption and integrity protection can be done at several gigabits per second without hardware support on 2008-era hardware. The newest Intel CPUs incorporating AES instructions are even faster—tcpcrypt can reach 9Gb/s using AES-UMAC on a dual-core i5 desktop, suggesting that six-core i5 servers should handle 10Gb/s. These results suggest that tcpcrypt should have a negligible impact on the vast majority of applications.

The main contribution of this work is not performance, though this is a prerequisite. There are no new cryptographic primitives, nor is the protocol especially novel. The main contribution is from putting well-understood components together in the right way to permit rapid and universal deployment of opportunistic encryption, and then providing the right hooks to encourage innovation and deployment of much better and more appropriate application-level authentication. This ability to integrate transport-layer security with application-level authentication largely obviates the need for applications to encrypt their own network traffic, thereby minimizing duplication of functionality.

As an example, we showed how a simple batch-signing server-authentication scheme can leverage tcpcrypt to provide forward secrecy and the same security as SSL while handling 25 times the connections per second. At the same time, the protocol allows an SSL server to fall back gracefully to regular SSL behavior when one or the other side cannot utilize tcpcrypt for authentication.

We also demonstrated the use of tcpcrypt to bootstrap both weak and strong password-based mutual authentication (using CMAC and PAKE respectively). Password-based authentication without mutual authentication, even over SSL, really should be a thing of the past. Using tcpcrypt with batch signing and CMAC mutual authentication is strictly stronger than HTTP Digest authentication over an SSL session, and more than 20 times faster. Using tcpcrypt and our unoptimized PAKE implementation is almost twice as fast as SSL, and provides stronger security. Many other authentication mechanisms are possible; we believe that tcpcrypt's generality and simple application-level hooks are exactly what is required to get application writers to think about the form of authentication they really need, once they can address authentication separately from the question of how to encrypt session data.

Finally, tcpcrypt interoperates seamlessly with legacy applications, TCP stacks, and middleboxes, making it easy to deploy incrementally. For all of the above reasons, we believe that it now makes sense to make transport-layer encryption the default. Make it happen by installing tcpcrypt from `http://tcpcrypt.org`.

Acknowledgments

We thank Alan Eustace, Daniel Giffin, Eric Grosse, Brad Karp, Adam Langley, the anonymous reviewers, and our shepherd, Nikita Borisov, for information, suggestions, feedback, and other assistance. This work was funded by gifts from Intel (to Brad Karp) and from Google, by NSF awards CNS-0716806 (A Clean-Slate Infrastructure for Information Flow Control) and CCR-0331542 (PORTIA), and by the EU FP7 Trilogy project.

References

[1] VIA Padlock Security Engine.

[2] Ultra-FAST Cryptography on the Sun UltraSPARC T2. `http://blogs.sun.com/bmseer/entry/ultra_fast_cryptography_on_the`.

[3] BITTAU, A., HANDLEY, M., AND LACKEY, J. The final nail in WEP's coffin. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 386–400.

[4] BONEH, D., AND SHOUP, V. A graduate course in applied cryptography. Version 0.1, from `http://cryptobook.net`, 2008.

[5] FEDERAL COMMUNICATIONS COMMISSION. Commission orders Comcast to end discriminatory network management practices. `http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-284286A1.pdf`.

[6] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWARD, L. HTTP authentication: Basic and digest access authentication. RFC 2617, 1999.

[7] GRANBOULAN, L. How to repair ESIGN. In *Security in Computer Networks* (2003), vol. 2576 of *LNCS*, pp. 234–240.

[8] GUERON, S. Intel Advanced Encryption Standard (AES) Instructions Set. Intel White Paper, Rev 03.

[9] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 9–9.

[10] HANSTEEN, P. N. M. *The Book of PF - A No-Nonsense Guide to the OpenBSD Firewall*. No Starch Press, 2007.

[11] HEFFERNAN, A. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385, 1998.

[12] HURLEY, C. Y,000,000,000utube. The Official YouTube Blog, `http://youtube-global.blogspot.com/2009/10/y000000000utube.html`.

[13] JACOBSON, V., BRADEN, R., AND BORMAN, D. TCP extensions for high performance. RFC 1323, 1992.

[14] JONCHERAY, L. A simple active attack against tcp. In *SSYM'95: Proceedings of the 5th conference on USENIX UNIX Security Symposium* (Berkeley, CA, USA, 1995), USENIX Association.

[15] KATZ, J., AND LINDELL, A. Y. Aggregate message authentication codes. In *Topics in Cryptology – CT-RSA* (2008).

[16] KENT, S., AND ATKINSON, R. Security Architecture for the Internet Protocol. RFC 2401, 1998.

[17] LANGLEY, A. Obfuscated TCP. `http://code.google.com/p/obstcp/wiki/Transcript`.

[18] LIN, C.-C. RSA Performance of Sun Fire T2000. `http://blogs.sun.com/chichang1/entry/rsa_performance_of_sun_fire`.

[19] MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. TCP selective acknowledgement options. RFC 2018, 1996.

[20] MCCARTHY, C. Pingdom: Facebook is killing it on page views. CNET News, `http://news.cnet.com/8301-13577_3-10428394-36.html`.

[21] PETER WALTENBERG. AES-GCM, AES-CCM, CMAC updated for OpenSSL 1.0 beta 2 – revised.

[22] RESCORLA, E. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000.

[23] RICHARDSON, M., AND REDELMEIER, D. Opportunistic Encryption using the Internet Key Exchange (IKE). RFC 4322 (Informational), December 2005.

[24] ROTHSCHILD, J. High performance at massive scale - lessons learned at Facebook. Seminar at UCSD.

[25] SCHILLACE, S. Default HTTP access for Gmail. `http://gmailblog.blogspot.com/2010/01/default-https-access-for-gmail.html`.

[26] TOUCH, J., BLACK, D., AND WANG, Y. Problem and Applicability Statement for Better-Than-Nothing Security (BTNS). RFC 5387 (Informational), November 2008.

[27] TOUCH, J., MANKIN, A., AND BONICA, R. The TCP authentication option. Internet draft (work in progress), July 2009.