Performance Characteristics of Application-level Security Protocols

Peter Gutmann
University of Auckland

Note: This draft is a work in progress and hasn't been officially published yet.

Abstract

Comparisons of the most popular application-level security protocols, PGP and S/MIME for independent message protection and SSH and SSL/TLS for communications session protection, are usually made at the political rather than the technical level. This paper provides a detailed breakdown and analysis of the performance characteristics of the different protocols, identifying potential performance problem areas and providing guidance for protocol designers and implementers.

1. Introduction

Over the years, a variety of application-level security protocols have appeared, of which the most common are PGP and S/MIME for message security and SSL/TLS and SSH for communications session security. Although the protocols are occasionally compared (usually through the discussion medium known as the flamewar), there don't appear to be any detailed technical analyses and comparisons of the various features and performance characteristics of the two pairs of protocols. The only other works in this area compare the OpenBSD IPsec and unspecified TLS and SSH implementations (presumably OpenSSL and OpenSSH) by benchmarking file transfer times relative to their unencrypted counterparts [1], and analyse a trace-driven simulation of a TLS server handling the workload of a typical e-commerce site [2], although timings for selected portions of security protocols may occasionally be found in other papers, typically when they're being used for timing-based attacks on the protocol [3].

This work in contrast looks at the impact of the design features (and occasional misfeatures) of the different protocols on both performance and message size, and examines hardware-specific issues such as performance on devices with limited resources and interaction with crypto hardware. The intent is to provide a technical basis for discussions about the different protocols' relative merits, to provide guidance to allow potential users to select the one that best suits their needs, and to analyse performance and operational issues that affect implementations of individual protocols.

Because of space constraints, this paper contains general summaries of a number of the findings. An extended version of the paper containing further test results and more detailed analyses will be made available on the author's home page.

1.1 The Testbed

In order to compare this range of protocols, it's necessary to have a unified testbed that runs all four protocols in the same environment, to avoid unfairly biasing the results in favour of a particular highly-tuned implementation of one of the protocols. This requires that all of the protocols run on the same hardware, use the same crypto algorithm implementations, have the same interface at both the front-end (for example the way in which messages are marshalled for processing by the crypto layer) and at the back-end (for example the networking layer handling for SSH and TLS). In addition it would be beneficial if the underlying crypto algorithm implementations were representative of ones that are widely used in practice so as to provide a good indication of the relative real-world performance of the different crypto mechanism/algorithm types.

The symmetric crypto and hash algorithm implementations used are libdes et al, originally written by Eric Young and made available under a BSD-style license[1], which are extremely widely used due to their adoption, via OpenSSL and BSAFE, in a large range of applications and security products, both open-source and commercial. The AES code is the standard reference implementation by Brian Gladman made available under an open-source

[1] This paper includes mention of cryptographic software written by Eric Young.

license, which is again very widely used. On x86 systems these include hand-tuned assembly-language implementations of the most common algorithms and modes that offer extremely good performance.

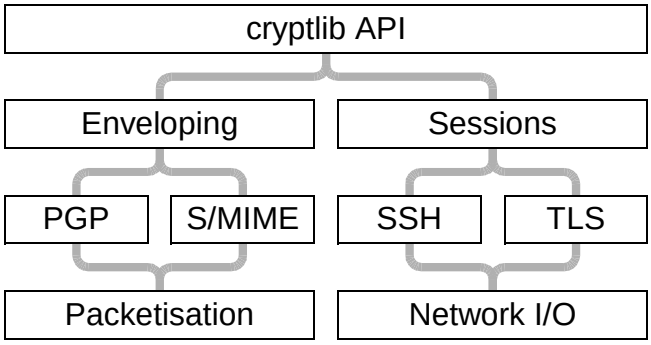

Figure 1: The testbed architecture

The all-in-one protocol implementation used to ensure consistency across the different protocols was cryptlib [4][5], whose architecture is shown in Figure 1. Both the secure messaging ("enveloping" in cryptlib terminology) and secure session interfaces share the same front-end and back-end code, with a plug-in intermediate layer adding the necessary wrappers to the data to distinguish PGP from S/MIME and SSH from TLS. Because they share the same code, the chances of implementation bias unfairly affecting the performance of one or more of the protocols being evaluated is small.

1.2 Performance Timing

Obtaining precise running times for operations on a multitasking, multithreaded system can be somewhat tricky due to a combination of lack of high-precision timers and interference from other processes and threads. The timing tests reported here were run under Windows, mostly because that's the only platform for which drivers for many of the crypto devices used are available, although for the software-only crypto a consistency check run under Linux and FreeBSD provided equivalent results to the Windows ones. Although Windows provides per-thread execution times using the `GetThreadTimes()` function, this has a granularity of 10ms (the OS scheduling time-slice) which is completely inadequate for timing operations that complete in a fraction of a millisecond. There are also several high-resolution timers available, either in the form of the `QueryPerformanceCounter()` function or the x86 `rdtsc` (read timestamp counter) instruction. On machines with a uniprocessor HAL (hardware abstraction layer) `QueryPerformanceCounter()` uses a 3.579545 MHz hardware timer, and on machines with a multiprocessor or APIC HAL it uses the CPU's TSC (timestamp counter). This choice of time sources is somewhat peculiar because on a multiprocessor machine it's theoretically possible to get completely different TSC readings depending on which CPU the thread is scheduled on, while for uniprocessor machines this isn't a problem. However, testing on two different dual-processor machines indicates that the kernel appears to synchronise the TSCs across CPUs at boot time (it resets the TSC as part of its system initialisation), so using the TSC shouldn't be a problem even on multiprocessor machines. Another quirk of the TSC is that mobile CPUs will turn it off when they idle, Pentiums with thermal management will change the rate of the counter when they clock-throttle (to match the current CPU speed), and hyper-threading Pentiums will turn it off when both threads are idle since the CPU will be in the halted state and not executing any instructions to count, all of which make reliable timing of crypto hardware operations using the TSC difficult. To avoid any ambiguities and keep things simple, we use `QueryPerformanceCounter()` and ensure that the timing tests are only run on machines with a uniprocessor HAL, in this case a 1.8 GHz Celeron.

A problem with using any high-resolution hardware timer is that it's affected by noise from other operations being performed by the OS or other applications that may be running. There are two ways of resolving this problem, the first being to apply standard statistical techniques to clean up the timing data and the second being to take the fastest result obtained, on the basis that this will be the one least affected by other operations (a background context switch or interrupt can only ever make things take longer, never speed them up).

The general concept behind the former method is to take the mean and then discard statistical outliers too far removed from it, sometimes referred to as using resistant methods to collect data. In this case what we're doing is fairly straightforward since we're measuring a completely deterministic process, and all we need to do is remove

sampling noise. As a general observation on the raw timing results, when graphed they represent a more or less straight line at, or close to, the minimum time, with the occasional spike several orders of magnitude out when a context switch occurs. Cleaning up the timing data involves removing these (very obvious) singularities to leave the actual results of interest.

The results from this process were tallied and compared to the alternative results obtained by taking the fastest possible time. In all cases they were within 0.5 – 1% of each other, showing that either method produces a reasonably accurate estimate of the relative performance of different crypto mechanisms even in the presence of background activity such as context switches and interrupt processing.

1.3 What was Measured

Evaluating the performance of a high-level crypto protocol turns out to be surprisingly tricky due to the number of external non-protocol related factors that affect it. The most obvious one is network and network stack performance for the secure communications session protocols. With a fast enough processor and/or a slow enough link and network stack, the network time will dominate the overall performance. In order to eliminate this bias, we exclude network I/O times from our measurements (these times can easily be determined for an individual target environment using standard tools like `ping` and `tcpdump`).

Other, less obvious factors can also affect the performance of a protocol. For example generating a DSA signature requires a source of high-quality cryptographic random numbers, while generating an RSA signature doesn't. This means that, all other things being equal, generating a DSA signature will always be slower than generating an equivalent RSA one because of the extra time added by the random number generator (RNG). In addition, the amount of extra time will vary based on the RNG used. A fairly simple (and relatively insecure) generator like the FIPS 186 one [6] will add very little overhead, while a high-security one containing multiple levels of processing and in-line self-checking such as the FIPS 140 tests [7] can add quite a bit of overhead. In addition some generators like the widely-used `/dev/random` [8], which perform entropy quality estimation, will block until enough entropy becomes available for them to continue. To eliminate this source of bias, we exclude the overhead added by the RNG (this approach has also been used by other researchers [2]).

Another potential source of measurement problems is the use of long-term vs. short-term keys. Since tampering with public key values allows an attacker to imbue them with all manner of supplementary properties not foreseen (or desired) by the original owner (the attacker's goal being a reduction or even nullification of security), most recent crypto implementations perform extensive checks on public/private keys when they load them in order to verify their validity, although some still don't, accepting keys with nonsensical values that provide no security [9]. The overhead added by this checking makes the key load rather expensive, which isn't a problem for (say) an SSL/TLS server that loads its private key once when it starts up, but can become expensive on the SSH equivalent using ephemeral DH keys (most recent SSH implementations are moving towards the use of this mechanism), which can potentially require a new key load for each connection.

Acquiring and verifying the keys and certificates needed for signature checking can also greatly influence the performance of a protocol. For example the time required to import and verify certificates in an S/MIME cert chain or to search a PGP keyring for trusted signers and import their keys can completely overwhelm the (relatively quick) signature verification operation. This is a protocol-specific issue covered in more detail in section 3.2. An algorithm-specific issue is the bulk data processing rate, which is purely a function of the underlying hash/encryption algorithm and will be identical for the two pairs of protocols. Because of this we don't analyse it here, algorithm-specific throughput is covered by a number of other sources, for example [1].

Finally, a small amount of situation-specific bias will affect any performance results. For example for the quickest operation we examine, PGP- and S/MIME-wrapping of plain data, around one third of the total processing time was consumed by the `malloc` call that allocated the working buffer to contain the data (Windows memory allocation can be rather inefficient, something examined in more detail in the extended version of the paper). Since one of cryptlib's targets is embedded systems that may have no dynamic memory allocation, it tends to allocate a large single block of memory and then sub-allocate portions of it for individual pieces of data, rather than calling `malloc` for each individual item of data in a PGP key or X.509 certificate, which makes it relatively insensitive to such issues. A more conventional implementation that allocates memory as required for each individual data item however may find itself malloc-bound. Although some care has been taken to avoid unfair bias in favour of a particular protocol, the reader is cautioned that some variance can be expected across different implementations and

operating environments. The sections that cover individual protocols contain more details on issues that can affect performance.

2. Protocol Versions and Variations

All of the protocols discussed here come in a variety of versions, and each version allows for a sometimes bewildering array of crypto mechanisms and message contents. Fortunately (at least for our analysis) only a small number of these many variations are actually used, so we can restrict ourselves to examining only the ones in common use. Beyond the protocol-specific mechanisms, all of the protocols allow for a set of symmetric encryption and hash algorithms, of which again only a small number see any real use. Everything uses at least triple DES (3DES) and SHA-1, with some use of AES in newer implementations. In addition there is some use of older or alternative algorithms for historical reasons (RC4 in SSL/TLS, IDEA in older PGP, and MD5 in S/MIME, older PGP, and SSL/TLS). The analysis presented here is restricted to the most widely-used algorithms and mechanisms (triple DES, SHA-1) to avoid overloading the reader (and author) with data for large numbers of relatively obscure and rarely-used algorithms. Performance data for individual algorithms is available from a variety of sources, including [1].

2.1 PGP

PGP comes in two versions, PGP 2.x (sometimes referred to as PGP classic) and OpenPGP [10][11][2]. PGP 2.x is deprecated but still enjoys some popularity because of the open availability of its source code, because support for it is built into a lot of existing software, and because it's seen as good enough by many users who see no pressing need to upgrade. In general though the more modern OpenPGP format is the preferred one, and because it's functionally (if not bits-on-the-wire) identical to PGP 2.x in most cases, only OpenPGP is considered here. Where the term PGP is used in the text, the OpenPGP format should be assumed unless explicit reference to PGP 2.x is made, with Elgamal key exchange and DSA signatures (OpenPGP) or RSA key exchange and signatures (PGP 2.x).

2.2 S/MIME

S/MIME also comes in a variety of versions that differ mostly in name. S/MIME is something of a misnomer since it merely refers to a MIME wrapping of an underlying cryptographic message format, and it's the underlying format that contains the security mechanisms. The original S/MIME was the PKCS #7 cryptographic message format [12] wrapped in MIME [13][14]. When the IETF took over PKCS #7 from RSA Data Security, the format was renamed Cryptographic Message Syntax (CMS) [15][16], with the result when wrapped in MIME again being called S/MIME [17][18]. CMS added a few (rarely-used) extensions to PKCS #7 and cleaned up some minor problems, but by and large the version that's normally used is identical to PKCS #7. The term S/MIME is used in the text to refer to PKCS #7/CMS with RSA key exchange and RSA/DSA signatures, the only version supported by most implementations.

2.3 SSL/TLS

SSL comes in two major and one minor versions. SSL (a.k.a. SSLv3) [19] and TLS (a.k.a. SSLv3.1) [20] are identical save for a few minor protocol details and a switch from the proto-HMAC (hashed message authentication code) used in SSLv3 (HMAC wasn't around at the time) to the actual HMAC in TLS, along with accompanying minor changes in the way the HMAC was used. TLS 1.1 (a.k.a. SSLv3.2) [21] is a minor tweak of TLS to include explicit initialisation vectors (IVs) in messages to counter a (somewhat unlikely) attack against SSLv3/TLS 1.0, which used the last block of the preceding packet as the IV [22][23]. The term TLS is used in the text to refer to SSLv3 and TLS 1.0 unless explicit reference to TLS 1.1 is made.

TLS supports a wide range of security mechanisms (cipher suites in TLS terminology), but in practice the only one ever used is RSA key exchange with RSA (or very rarely DSA) signatures, a choice popularised originally by Netscape and then by Microsoft's Netscape-compatible SSL implementation. Most applications don't support any of the others, so we can restrict ourselves to examining only this option rather than the full fifty-four possible cipher suites.

[2] The use of ephemeral drafts as references is an unfortunate necessity. Like open source software, many Internet security protocols are in permanent beta.

2.4 SSH

SSH comes in two versions, 1 and 2, of which version 1 has mostly fallen out of use due to widely-publicised attacks against both the protocol and individual implementations. SSH version 2 is a complete re-design of the original [24][25][26][27] resulting in a protocol very similar to SSL when DH key agreement is used (cryptlib actually recycled the SSL protocol engine for its SSHv2 implementation). This similarity wasn't because of any deliberate attempt to copy SSL (quite the opposite), but because the most straightforward (and secure) design for a protocol of this type tends to end up looking like SSL/SSHv2 [28]. In order to simplify things in the analysis, the terminology pioneered by SSL is used to describe both protocols. The term SSH is used in the text to refer to version 2 of the SSH protocol.

3. Secure Messaging Protocols: PGP and S/MIME

PGP and S/MIME follow the general pattern shown in Figure 2 for signed and encrypted messages. For encrypted messages a key-encryption key (KEK) is used to wrap up a content-encryption key (CEK) and the resulting key exchange (keyex) information is prepended to the payload. The payload is then encrypted with the CEK using the algorithms covered in section 2. The recipient uses the KEK to decrypt the CEK in the key exchange information, and uses the CEK to decrypt the payload. There's really only one way to perform this kind of operation, so PGP and S/MIME encrypted messages differ only in their choice of bit-bagging formats.

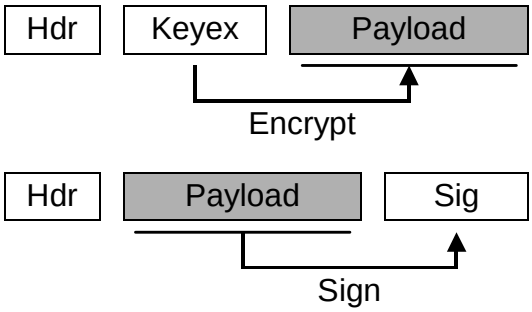

Figure 2: PGP and S/MIME message layout

For signed messages, the message payload is hashed and the resulting hash value signed with the sender's private key, as shown in Figure 2 and again using the algorithms from section 2. The recipient performs the same hashing using information about the hash algorithm contained in the message header, and checks the signature on the hash once it reaches the end of the message. As with encrypted messages, the reason why the PGP and S/MIME formats (at least at this abstract level) are identical is because there's really only one logical way to perform the required operations. Consider for example an implementation that places the signature before rather than after the payload. This makes a one-pass/streaming implementation impossible, requiring that the sender either make two passes over the data or that they buffer the entire message before encoding it, because they can't emit the payload until they've emitted the signature. PGP 2.x actually handled signed data in this way, requiring multiple passes over the message to be signed, with intermediate results spooled to disk before the next pass could begin [29]. This made one-pass signature generation impossible, a problem fixed in OpenPGP.

The same applies to other operations such as omitting the hash information in the signed message header. In this case it'd be necessary to read (and buffer) the entire message in order to get to the signature containing the hash information that's needed to process the message. In the case of encrypted messages, placing the encrypted CEK after the payload rather than before it would again entail buffering the entire message before processing could begin. The similarity between the (abstract) PGP and S/MIME messages isn't because of any deliberate attempt at copying (there was in fact some hostility between the two camps at the time the formats were being laid out), but because that's the most logical and straightforward way to do things.

When it comes to adding frills however, there is some leeway for personal preferences to creep into the design. Both PGP and S/MIME allow arbitrary additional attributes to be added to the signed data (PGP 2.x didn't allow any attributes other than a few fixed values to be added). This is done by extending the signature to cover the additional authenticated attributes. Both formats also support unauthenticated attributes, data included in the overall message

but not covered by the message payload signature. These are used to provide facilities such as countersignatures and timestamps on the existing signature.

The form chosen by S/MIME is to include the hash of the payload as another attribute in the authenticated attributes, and then to sign the attributes, signing the message with one level of indirection as shown in Figure 3. This is a simple, elegant mechanism that allows some flexibility in signature creation, allowing for example detached signatures in which the payload is separated from the signature data without any need to change the signature format. This is used in mechanisms such as Microsoft's AuthentiCode, in which a detached signature is embedded as a COFF record in an executable, with the signature covering the actual executable code (alongside portions of the COFF header) [30][31].

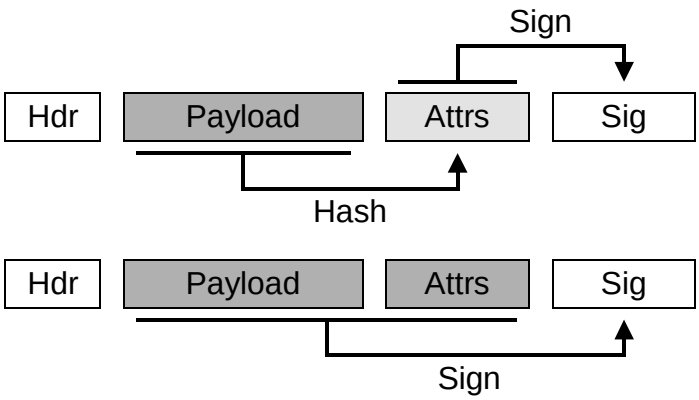

Figure 3: S/MIME (top) and PGP (bottom) signed attribute handling

PGP in contrast takes a slightly different approach, extending the signature to also include the authenticated attributes (this was done as a size optimisation to save 20-30 bytes in the original PGP design, which was very size-conscious [32][33]), as well as some additional implicitly-included data not shown in the diagram whose purpose is no longer known but was probably a leftover from an early PGP design [34]. This format, although somewhat simpler than the S/MIME approach, is also less flexible, since the payload has been extended to include the authenticated attributes. As a result it's no longer possible to verify the attributes by themselves, or to create a timestamp or countersignature simply by countersigning the attributes.

3.1 Message Size

By and large (and depending on how many frills are added), there isn't much difference between PGP and S/MIME message sizes. Table 1 shows the overhead (not counting the data payload size) when enveloping a short test message as plain data, password-encrypted data, public-key encrypted data, and signed data, with a minimum of frills added (in other words the data format has been chosen to minimise message size). Public keys were 1024 bits in length, with RSA adding $^{bits}/_{8}$ bytes, Elgamal $^{bits}/_{4}$ bytes, and DSA 0 bytes for every extra bit of key length over 1024.

	PGP		S/MIME	
Plain data		8		17
Password-encrypted		35		168
Public-key encrypted	(RSA) (Elg.)	162 293	(RSA)	235
Signed	(RSA) (DSA)	181 95	(RSA) (DSA)	238 112

Table 1: PGP and S/MIME message overhead in bytes

As the table indicates, the difference between the two is mostly constant, and is chiefly due to the fact that S/MIME has a more flexible format than PGP while PGP prefers to optimise for size, which results in a slightly larger encoded S/MIME message size. For example PGP uses one-byte algorithm identifiers specified in the PGP RFC, while S/MIME uses multi-byte universal object identifiers (globally unique IDs) capable of encoding any type of algorithm along with any optional parameters that the algorithm may require. This extra flexibility comes at a slight cost in message size. In terms of algorithm-specific quirks, DSA generates fixed-size signatures (2×160 bits) and Elgamal requires two data values to be exchanged in place of RSA's one, so it grows at twice the rate. Both S/MIME and PGP use a compact, variable-length encoding of message length information, so the overheads indicated in Table 1 remain relatively constant over different message sizes, changing by only a few bytes as the length encoding changes with size.

The reason for the noticeable size difference in the password-encrypted data is because of the different way that this is handled by S/MIME and PGP. S/MIME treats the password as a standard KEK, so that the encryption process works exactly as it does for public-key encryption. PGP however directly derives the CEK from the password, which is slightly more compact but makes it impossible to encrypt a message for more than one recipient, or to allow mixed public-key and password-based encryption for the same message.

3.1.1 S/MIME Signatures

S/MIME has a reputation for extremely large signed messages, which has two causes: signing keys are identified using an unwieldy and awkward variable-length identifier, and signatures traditionally include with them all of the certificates necessary to verify the signature. The unwieldy identifiers are required because the traditional way to identify a certificate is through the encoded form of its X.500 distinguished name (DN) combined with the certificate serial number to uniquely identify the individual certificate that was used. In practice it's treated as an opaque blob for which the only valid operation is a comparison for equality, but the fact that it's a variable-length blob makes it consume more space than necessary (a number of implementations convert it to a fixed-size hash internally for ease of use). A survey of several hundred certificates pulled from the online repositories of a number of public CAs and the author's own collection (known in PKI circles as the "certificate zoo" because of the many weird and wonderful denizens that inhabit it, although some would be more at home in Ripley's Believe-it-or-not) revealed an average encoded DN size of 140 bytes, for a total encoded DN and serial number average size of 150 bytes. PGP in comparison uses a fixed-size 64-bit hash of key components and a few incidentals to identify keys.

CMS introduced a new way of identifying certificates to supplement the original PKCS #7 DN and serial number, the key identifier. This consists of a short binary string uniquely tied to a key or certificate (it's embedded in the certificate as an attribute) that can be used to locate the certificate or key needed to verify the signature on, or decrypt, a message. Unfortunately the PKIX standards group (which specifies Internet PKI standards) and the S/MIME standards group disagree as to whether the key identifier should be unique or not: S/MIME says it should be, PKIX says it shouldn't, which makes it of little use for uniquely identifying keys [35]. To make things worse, the PKIX core certificate RFC recommends using monotonically increasing integers as one of its two alternatives for generating key IDs, which practically guarantees non-unique IDs (the other alternative also generates non-unique IDs, although with a much smaller likelihood). A short-term (but rather unsatisfactory) workaround to this problem is to either use only products from vendors that use (probabilistically) unique PGP-style hashes of key components, or to reject any key IDs of less than (say) 40 bits as being unlikely to be unique. A better long-term solution would be for S/MIME to allow a hash of the certificate (already in almost universal use as the "fingerprint" or "thumbprint") as a unique key identifier.

The second size problem with S/MIME messages arises because of the inclusion of signing certificate(s) with the message's signature. This practice arose because of the lack of a PKI of the kind originally envisaged in X.500 to obtain the certificates, and will likely continue into the foreseeable future. Because certificates are relatively large (1KB or more) and there are usually several of them attached to a signature, the size of these signatures when sent in signed email is frequently much larger than the message itself (as well as being a rather intrusive block of base64-encoded text in non S/MIME-aware mailers). A sample of S/MIME signed messages gathered from a variety of mailing lists and representing a fairly broad cross-section of both S/MIME applications and CAs indicates an average size of around 2.4KB for a typical signature. Since the default format for S/MIME signatures uses both the variable-length DN and certificate serial number to identify keys and attaches certificate chains for signatures, these values need to be added to the figures in Table 1 to obtain the worse-case size when the format used isn't selected to minimise the overall message size.

3.1.2 Authenticated Attributes

As was mentioned earlier, both PGP and S/MIME allow the use of authenticated attributes, additional information included with the signed message payload. Although it's possible to include an arbitrary number of these attributes (a small number of standardised ones and an arbitrary number of user-defined or application-specific ones), both formats only specify a small number of default attributes. These include the signing time, and frequently an indication of the signed message type to avoid situations where an attacker could take a signed message from protocol A and insert it into protocol B, where it would have an entirely different meaning. Other standardised attributes include various security-related parameters such as encryption algorithm preferences, signer/sender identity information, meta-data tied to the message payload, and anything else the standards authors though might some day be useful.

The typical encoded size for the PGP default attributes is 20 bytes, for the S/MIME defaults it's 90 bytes (however, in PGP they're mandatory while they're optional in S/MIME). Other attributes that might be found in an S/MIME signature are an indication of the signer's security mechanism preferences when receiving messages, in the range of 50-150 bytes, and oddities such as a duplicate copy of the certificate identifier added by Microsoft applications, perhaps as a spare in case the other one breaks down. Like S/MIME, PGP also includes a means of specifying the security preferences, but places this in a more logical location attached to the public key rather than including it in signed messages. The reason for this is that PGP users are allowed to sign their own key data and can include/update their preferences as required, while S/MIME users have to have their keys signed by a CA, so the only user-signed location where they can specify their preference is as part of a signed message.

3.2 Message Processing Overhead

As with message size, there isn't that much difference in processing overhead between S/MIME and PGP. Table 2 extends Table 1 to show the message-processing overhead for the various message types. As was mentioned earlier, since both protocols are using the same crypto algorithms and mechanisms, the speed ratios will remain constant as different algorithms are substituted.

	PGP	S/MIME
Plain data	.19	.25
Password-encrypted	.92	2.20
Public-key encrypted	8.03	6.25
Signed	11.84	13.87

Table 2: PGP and S/MIME message processing overhead in milliseconds

There are few surprises here. Apart from the bit-bagging format used, what PGP and S/MIME do are pretty much identical, so there shouldn't be any significant differences between the two. The one obvious difference is for the password-based encryption, where the cost of S/MIME's increased flexibility via an extra level of indirection shows: There's extra overhead added by wrapping the CEK in the KEK, while for PGP the password is converted directly into the CEK. In addition for public-key encrypted enveloping PGP's key wrapping is slightly more complex than S/MIME's (it performs extra processing of the data being wrapped), and it uses the somewhat less common CFB encryption mode rather than the more widely-used CBC that S/MIME shares with TLS, SSH, and IPsec, which doesn't have an assembly-language implementation in libdes and is therefore slightly slower. As a result, PGP public-key encrypted enveloping takes slightly longer than the S/MIME equivalent.

The figures change somewhat when we include the key management overhead for signing already discussed in section 1.3. For S/MIME certificate chains, each certificate is a complex, composite structure that requires considerable parsing and processing, and each chain consists of a number of these certificates. Providing absolute figures for the overhead added by this is difficult since it depends greatly on the certificate chain being processed. Actual certificate chain import time ranged from 1.5ms (amazon.com's certificates, a standard example of a Verisign-issued certificate chain, from the TLS tests in section 4.2) to 95ms (a large chain containing extremely complex certificates from a European CA). Verifying the certificates in the chain added another 2-3ms.

In practice these figures will vary enormously, and a worst-case example chain that takes 7 times as long to process as the signed data should not be taken as representative of general performance. In this instance cryptlib is at the high end of the scale since it performs a full check of all certificates in the chain against the PKIX specification, leading to a relatively high overhead for certificate chain processing. Other implementations may perform little or no checking, accepting broken or invalid certificates but being quite efficient at doing so. Because of these differences, there'll be quite a range of performance figures for this part of the operation. If performance is a concern, it's possible to speed up this step by using an existing copy of the certificate to verify the signature rather than re-creating it each time from certificate data included with the message. Most implementations provide for this in the form of a certificate trust database containing known-good certificates trusted by the user and/or application. The figures given above are strictly worst-case (or at least non-optimal) performance indicators.

For PGP the verification time isn't anywhere near as simple to quantify. Since PGP doesn't communicate signing keys with the message, an external lookup is required to locate the key, and like S/MIME certificates PGP keys are complex composite structures requiring a fair bit of parsing and processing. In addition if the web of trust mechanism is used, an arbitrarily large number of further lookups may be necessary to locate the remaining keys necessary to verify the signing key, and since these lookups involve expensive PGP keyring searches or web accesses rather than following an in-memory certificate chain, providing timing figures for these operations would be somewhat misleading. The principal difference then between the S/MIME strict hierarchy and the PGP web of trust is that the hierarchical verification time is bounded while the web of trust time is potentially unbounded, although in practice the webs don't get very large so this doesn't become a real problem. As with S/MIME, PGP can also employ trust databases to speed up this verification process, providing the same performance as an equivalent cert trust database.

A similar situation applies for decryption, where the time required to locate and read a private key totally overwhelms the decryption time, not helped by the fact that private keys in software-only implementations are stored in encrypted form with many iterations of processing designed to slow down password-guessing attacks. As with key trust databases, caching the private key in memory can eliminate this extra overhead, although this must be balanced against the risk of having the private key sitting in memory in plaintext form [36]. Because of this highly variable time for de-enveloping signed and encrypted data, Table 2 only includes times for enveloping the data, although an indication of the relative speed of the corresponding public/private-key operations used in de-enveloping can be obtained from the algorithm speed ratios in section 4.2.

Signed attributes add an essentially negligible time to the overall message processing time, with S/MIME being slightly slower then PGP, mostly due to the very slight overhead of creating a new hash object to handle the separate hashing of message body and authenticated attributes.

From these figures PGP fans may claim that S/MIME is 20% larger than PGP, which is clearly excessive. S/MIME advocates on the other hand may claim that it only requires 25 more bytes than PGP, which is obviously negligible. It's important to remember though that the real overhead is minimal in either case — a typical encrypted/signed email message has (excluding base64-encoding) an overhead of 100-200 bytes added by the security processing, about the same as a line or two of SMTP `Received:` headers.

4. Secure Communications Protocols: TLS and SSH

The TLS and SSH protocols follow the same pattern: Both sides announce their encryption capabilities and agree on cipher suites (bundles of encryption, authentication, and digital signature algorithms) to use, they perform a key exchange using the agreed-upon cipher suite, use the single master key resulting from this to derive individual keys for encryption, authentication, and other purposes, and finally they send a change cipher-spec message to the other side to indicate a switch to the newly-negotiated cipher suite. From then on the exchange continues in secure mode. At this point TLS is done, and SSH continues with further authentication negotiations and other housekeeping.

When either side wants to terminate the session, both sides exchange close notification messages before closing the session to detect truncation attacks by outsiders forcing a shutdown by forging TCP FINs. The message flow for both protocols is shown in Figure 4.

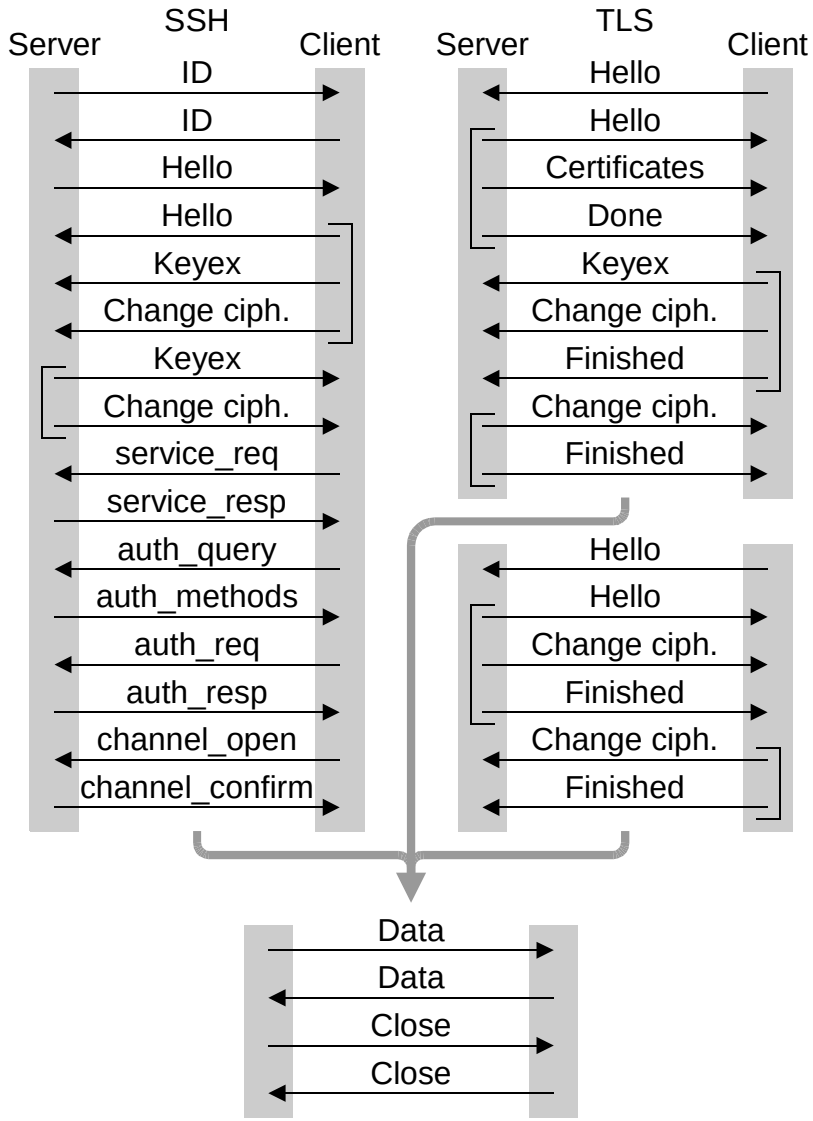

Figure 4: SSH (left), TLS (right top) and TLS-sharedkey (right bottom) message flow

There are some differences between the TLS and SSH exchanges. The main one obvious from Figure 4 is that SSH uses key exchange messages from both sides while TLS only has a key exchange message sent by the client. This is dictated by the choice of key exchange algorithms, TLS uses RSA which requires a message only from the client while SSH uses DH which requires messages from both client and server. If TLS is used with DH (which it almost never is, see the comment in section 2), the exchange looks a lot like the SSH one.

Another difference is that TLS sends a finished message during the initial handshake while SSH doesn't. The finished message contains a cryptographic checksum (MAC) of all messages exchanged so far, using two different algorithms (MD5 and SHA-1) for redundancy in case one of the two is ever broken. This serves two purposes, it verifies the integrity of all data in the handshake process (an attacker can't modify messages sent or received by either side without it being detected), and it proves to both sides that the peer is in possession of the correct cryptographic keys to continue the exchange. If a single bit in a handshake message has been altered, or a single key bit incorrectly computed, the finished message will detect it before any sensitive data is exchanged.

SSH uses a slightly weaker mechanism in which the server hashes portions of the handshake messages and then signs the result, with the client verifying the signature. This protects the portions of the handshake that are covered

by the hashing, but doesn't provide mutual proof of possession of the correct keys until further messages (potentially containing sensitive data) have been exchanged.

A final difference, not shown in the simplified diagram of Figure 4, is that TLS performs a handshake while SSH is more of a negotiation, with restarts and different message flows to the one shown in the diagram being possible. In TLS, the exchange is simple and straightforward: The client announces its capabilities, the server chooses what it deems most suitable and informs the client of its choice, and the client performs the key exchange. SSH in contrast has both sides shout at each other and then falls back to a complex interlock protocol (the description stretches over more than two pages) to sort out what happens next. Many clients wait for the server to speak first in order to avoid this issue. If there's a disagreement over how to proceed, both sides start again from scratch.

The situation is further complicated by the fact that the protocol allows speculative key exchange packets to be sent after the initial hello, which also have to be discarded if the shouting match fails to come to a clear conclusion. One vendor's implementation always opens the negotiation with a handshake using an under-specified mechanism whose format no-one can quite agree on, so the only appropriate response to a connect attempt from this particular implementation is to automatically discard all of the handshake messages and start again.

A final difference between the two protocols is the lack of a `Certificates` message sent by the SSH server. This is because the core SSH protocols use a fixed, hard-coded 1024-bit DH key borrowed from IPsec, so there's no need to communicate server key exchange material to the client. If per-server key exchange keys are used (most newer implementations do this), the SSH handshake is extended by a further two messages, one to request the key and one to return it, performing the equivalent of the TLS `Certificates` message [37].

4.1 Protocol Overhead

As Figure 4 shows, there's little difference (at an abstract level) between the initial handshake portions of the SSH and TLS protocols: TLS with public key-based key exchange takes two round-trip times (RTT) to complete its handshake and 1 RTT with password/shared-key exchange (the final client message can be combined with the first data packet). SSH requires 2.5 RTT without the use of a per-server key and 3.5 RTT with (an extra RTT is added for sending and receiving the server key). The real overhead for SSH occurs not with the initial handshake but in the post-handshake negotiation, which proceeds in a piecemeal manner for some time after the initial handshake has completed, so that a typical SSH handshake requires 6.5/7.5 RTT to complete. As section 4 indicated, there is quite a bit of variation in this part of the exchange, what's shown in Figure 4 is a representative example.

4.2 Message processing overhead

The times for each phase of the exchange (excluding supplementary overhead discussed in section 1.3) are shown in Table 3.

	SSH		**TLS**		**TLS-sharedkey**	
	Client	**Server**	**Client**	**Server**	**Client**	**Server**
Client hello	.04	.06	.05	.04	.04	.01
Server hello	.08	.11	<.01	.08	<.01	.06
Client keyex	23.18	36.08	1.40	10.54	—	—
Server keyex	15.79	34.09	—	—	—	—
Change ciph.	.48	.48	.80	.81	.81	.82

Table 3: SSH and TLS message processing overhead in milliseconds

As the timings show, SSH has a considerably higher cryptographic overhead than TLS, requiring a public-key DSA (or RSA) operation and a (pseudo-) private-key DH operation on the client and a private-key DSA (or RSA) operation and (pseudo-) private-key DH operation on the server. TLS in contrast requires only a public-key RSA operation on the client and a private-key RSA operation on the server, or no public-key operations at all if shared-secret/password-based key exchange is used.

Another problem with SSH's high-overhead key exchange can be seen from packet traces of the SSH handshake. On many systems the cryptographic overhead is high enough that it triggers TCP's delayed-ACK mechanism, so that a number of additional minimum-length ACK packets are sent because the delayed-ACK timer expires before the crypto processing has completed. Initiating SSH and TLS connections between identical machines showed that in the TLS handshake the ACKs were successfully piggybacked on TLS response messages, while for the SSH connection the ACKs were sent as separate packets because the SSH responses weren't available in time to piggyback the ACKs (this and other networking issues are covered in more detail in the extended version of the paper available from the author's home page).

Looking at the crypto algorithm performance, the most obvious difference is the speed of the RSA vs. DH key exchange. RSA is inherently quicker than DH, and RSA public-key operations are far quicker than private-key ones. In addition, DSA signature and verification takes approximately the same amount of time, so there's no benefit to trying to move the quicker one to the side with the least CPU power (to make the comparison a bit fairer, the SSH test switched to the quicker RSA algorithm used by TLS instead of using the SSH mandatory DSA algorithm for signing). As a general rule of thumb, the ratio of times for the various public-/private-key timings relative to the quickest time are RSA public-key: 1, RSA private-key: 15, DSA public-key: 10, DSA private-key: 10, DH: 50. These ratios vary depending on the compiler and CPU type, for example the largest ratio, RSA public-key : DH, varied from around 1:40 to 1:60 over a collection of Intel PII, PIII, Pentium 4/Celeron, and AMD Duron, Athlon, and Athlon XP CPUs and steppings. These ratios illustrate the serious win obtained through the use of the Chinese remainder theorem (CRT) shortcut in the RSA private-key operation [38], which computes two $^{n}/_{2}$-bit modular exponentiations in place of DH's single *n*-bit one. Although two modular exponentiations are required instead of one, each is considerably more efficient because the values are smaller [39].

As for the non-cryptographic differences, TLS hello processing is slightly slower on the server because it has to update the TLS session cache (which allows quick session resumption), and SSH hello processing is slower than the TLS equivalent because although SSH uses binary data packets, it encodes algorithm choices in an awkward comma-separated ASCII text format that requires considerably more processing (and more careful processing to address potential malformed-packet attacks) than the TLS equivalent, a simple sequence of 16-bit integers (a detailed breakdown of TLS-specific protocol performance may be found in [2]).

4.3 Performance on Low-powered Devices

From their origins in web browsers and securing access to Unix workstations, TLS and SSH have ended up in some truly surprising places. Beyond the more obvious applications of securing access to web-enabled devices that employ HTTP control channels, TLS and SSH are used for secure financial data transfers, to provide secure firmware upload capabilities, to secure device-monitoring mechanisms, and in a variety of other applications. Because of this propagation of TLS and SSH into areas they were never really designed for, it's instructive to examine their performance, and the potential for performance optimisation, when used with low-powered embedded devices. In particular the high overhead of the crypto operations is a concern for low-powered devices such as embedded systems, whose CPUs can run at clock speeds as low as 33 or even 16 MHz. Timings for the crypto operations on common low-powered CPUs using 1024-bit keys are given in Table 4.

Beyond the relatively well-known ARM and embedded PowerPC cores, the table also covers the Hitachi SH3 and SH4 family (a 32-bit RISC CPU used in industrial and automotive controllers and handheld PCs), the AMD SC400 (an embedded SLE486 core) and SC520 (an embedded 5x86 core, in practice a 486-class CPU), the NS Geode (an embedded Pentium-class core), and the MIPS core (usually combined with various peripherals on a multifunction chip, and even available in open-source form) [40]. The hardware itself ran the gamut from handheld PCs to routers, wireless gateways, printers, cash registers, cameras, UPSes, industrial controllers, USB tokens, and a few other unusual and unexpected applications.

	RSA pub.	RSA priv.	DSA pub.	DSA priv.	DH
ARM7 33 MHz	TBA				
ARM7 66 MHz	TBA				
SH3 133 MHz	TBA				
SH4 180 MHz	TBA				
AMD SC400 99 MHz	21	399	206	211	1,178
AMD SC520 133 MHz	13	218	124	147	705
NS Geode 266 MHz	4	61	34	39	192
MIPS 300 MHz	TBA				
405GP ??? MHz	TBA				
405EP ??? MHz	TBA				

Table 4: Algorithm performance on embedded systems in milliseconds

Note: Some of these figures are currently still TBA, either because getting the hardware into a state where it's possible to perform timing measurements can be tricky (some of it involves attaching logic probes to I/O lines and timing the result with an external timer), or because publishing performance results for someone's hardware requires clearance from 15 levels of management. The final paper will contain complete figures.

The advantages of TLS' asymmetric key exchange when used with low-powered clients are very obvious from these figures. On the client, TLS requires a single, quick RSA public-key operation to encrypt the keying material being sent to the server, while SSH requires an expensive DH (pseudo-)private-key operation and a DSA or RSA signature verification. On the server side, TLS requires a single private-key operation (RSA decrypt) while SSH requires two (DH key agreement plus DSA or RSA signature). TLS also has a shared-secret handshake mechanism intended for use in situations where CPU cycles are at a premium (and as a side-effect provides mutual authentication of client and server without the use of certificates) [41]. This shared-secret mode eliminates the public or private-key operations on both sides, along with the certificate-processing overhead, with timing results as shown in Table 3.

Even when use of the shared-secret mode isn't possible, it's still possible to perform an extremely quick TLS connection by taking advantage of the asymmetric processing characteristics of the TLS handshake and having the embedded device perform a reverse connect to the controlling PC. In this case the PC connects to the device and indicates the desire to have the device initiate a communications session with it. The device then initiates a standard TLS session with the caller over the existing channel, with the (relatively) heavyweight private-key RSA operation performed on the controlling PC, and the device needing to perform only a lightweight RSA public-key operation. The only slight downside to this is the fact that the TLS pre-master secret (the source of further encryption keys) is now chosen by the embedded device, which will probably have a lower-quality source of randomness than the controlling PC (in at least one application this presented something of a problem until it was realised that the embedded controller had access to around half a megapixel of randomness updated several times a second, other embedded controllers can obtain similar input from the systems they control or monitor). This trick, which makes it possible to run TLS on the most underpowered or heavily CPU-loaded devices, isn't possible for SSH since it requires a heavyweight DH operation alongside the RSA public-key one.

5. Interaction with Encryption Hardware

Although the majority of the implementations of the protocols discussed here are in software, some are also implemented in hardware (or with hardware assist), either for performance reasons, because of security concerns over software-only implementations, as a transparent front-end to a non crypto-aware system, or for various other reasons. Hardware crypto implementations impose their own constraints that aren't present in software-only implementations. This section looks at some of these constraints, interactions with protocol features, and performance implications of both the hardware and the protocols.

5.1 Overview of Encryption Hardware Operation

For the purposes of this discussion we'll assume the use of a PCI-style hardware interface [42][43], which is almost always the case (there are more exotic crypto hardware interfaces such as HyperTransport and SPI-3/SPI-4.2 around, but they're only used for chip-to-chip interconnects). Some older devices use a SCSI or even a telnet-style interface over Ethernet (the latter functions as a crypto handbrake rather than an accelerator), but these have mostly fallen out of use except for specialised circumstances. PCI requires that all data transfers be 32-bit aligned (64-bit aligned for 64-bit PCI), with non-32-bit transfers being handled by asserting four (eight for 64-bit PCI) byte enable lines to select the data of interest. Data is transferred at the rate of 32 bits (64 bits for 64-bit PCI) per 33MHz (66MHz for 66MHz PCI) clock cycle, typically into and out of FIFOs on the crypto device that are used to match the device's I/O rate to the PCI bus rate. This results in a theoretical peak half-duplex throughput of 132/264/528 MB/s for standard/64-bit/64-bit and 66MHz PCI. The majority of add-in crypto hardware cards use standard 32-bit 33MHz PCI.

Data is transferred to/from the device using PCI read multiple (burst read) and write commands when the device uses (or emulates) memory-mapped I/O, but may have to be done via individual (and relatively high-overhead) read/write cycles for I/O ports, which cuts throughput by 75%. Most devices use memory-mapped I/O, allowing the use of burst reads and writes. Even then however, it's possible for a burst to be cut short by other bus activity, depending on the arbitration scheme used by the PCI bus arbiter. This can occur in high-performance server applications where the crypto device is sharing a PCI bus with network card(s) and disk drives, since each network packet will require bus access by both the NIC and the crypto device. This is why some vendors have tried to unload crypto processing into network hardware targeted for server use, rather than relying on separate crypto hardware on the host bus.

Some crypto hardware, particularly high-performance throughput-optimised units, don't store state in the crypto device but leave it to be managed by the caller. This means that it's not possible to interrupt and then restart a crypto operation to allow processing a message in two or more separate steps. For many application domains, including most types of IPsec processing (see the next section), this isn't a problem, and many devices have scatter/gather support to handle messages broken into parts and stored in several different locations. An example of this style of device is the Broadcom 582x series, frequently used in devices where high crypto performance is required. These take a control packet consisting of a command, a key (if required by the algorithm), and a set of memory locations for scatter/gather operations. The crypto operation is performed as a single step, and the data is written back to the output(s). Parameters such as crypto initialisation vectors (IVs) are handled by prepending them to the data or specifying them as data to be en/decrypted at a separate memory location, with the device itself being unaware of their existence as anything other than normal crypto data.

Even without other bus activity, a burst read or write can't continue indefinitely, being constrained by the internal storage available to the crypto device. These typically contain input and output FIFOs, with a burst write filling the input FIFO from which data is piped through the cryptologic into an output FIFO. Once enough data accumulates in the output FIFO, another burst write (this time from the crypto device, if it's bus-master capable) is initiated to transfer the data back to system memory. The total length of a burst transfer is therefore limited by the device FIFO depth, typically 64-256 bytes. The FIFOs act not just as a buffer for burst reads/writes but also provide an impedance-matching capability to match the cryptologic data rate to the PCI bus rate.

A final issue that affects crypto hardware is that of memory alignment. The need for 32-bit alignment of data quantities for optimal PCI bus transfers has already been mentioned, however there may be even more stringent page-alignment requirements in order to implement zero-copy I/O. In zero-copy I/O one or more pages of data are transferred across protection domains (for example from user space to the kernel) via the system's VM facilities, avoiding the need to copy data into kernel buffers before processing it. Zero-copy I/O was first implemented in the form of fbufs, an aggregate of one or more virtual memory pages accessed via scatter/gather mechanism for which a "copy" involves remapping the page(s) across protection domains, as opposed to the traditional mbufs, which require a real copying of data [44]. Similar mechanisms have since been implemented in a range of operating systems. In practice the operation is somewhat more complex than this, and many optimisations are used to cut down on overhead in manipulating the VM subsystem [45][46][47].

5.2 Interaction of Encryption Hardware with Security Protocols

When IPsec was designed, some consideration was made for hardware support (both crypto hardware and CPU-specific issues), although the final RFCs don't necessarily reflect this. The fact that the IP header is a multiple of 32 (IPv4) or 64 (IPv6) bits in length helped with memory alignment issues. In addition the ESP data is arranged to allow one-pass processing (AH isn't, since it requires that various bits and pieces that precede the AH be included in the AH processing, but this can be handled via scatter/gather if the crypto hardware supports it).

Non-IPsec protocols in contrast were designed purely with software implementation in mind. All of them use odd- or variable-length headers that make efficient processing using crypto hardware difficult. For the secure messaging protocols this isn't that much of a concern because they're rarely used in situations where hardware assist is required, and because their variable (and potentially huge) message lengths would make them difficult to process on hardware that requires single-step processing. If high throughput really is required (for example when authenticating or encrypting large numbers of small messages for some form of real-time transaction processing) it's possible to force alignment by taking advantage of the fact that although both PGP and S/MIME use variable-length encodings, they don't require minimal-length encodings, so that through the judicial application of leading zeroes when encoding length values (for example encoding a length of 128 bytes as `00 80` or `00 00 00 80` instead of `80`) it's possible to obtain any required memory alignment for the data that follows. For example the real-time certificate status protocol (RTCS) has been specially designed to allow for this type of processing, with fixed-length messages allowing the use of hardware assist to provide the maximum throughput, even though the underlying data format is variable-length [48].

SSH and TLS are more problematic. TLS uses an odd-length 5-byte header, which means that the data subject to security processing isn't properly aligned for hardware assist. Although it's possible to force the data to be aligned by storing the header as positions -5...-1 instead of 0...4, this now makes the overall packet non-aligned, which makes zero-copy network I/O impossible. Because of this misalignment, it's necessary to perform three PCI transfers to fetch the first two 32-bit blocks, and to repeat this process whenever a PCI transfer needs to be restarted due to a full device FIFO or termination of a burst transfer. Even a pure software implementation suffers slightly from this lack of alignment, although the slowdown is relatively small, only one or two percent over the range of x86 family CPUs mentioned in section 4.2.

SSH is slightly different. It encrypts the header along with the payload data, so that the packet is aligned for both crypto and network I/O. This convenience when encrypting comes at a high cost during decryption, however. Since the header is encrypted, it's not possible to know how long a packet is, and what to do with it, without decrypting at least some of the packet. As shown in Figure 5, processing an incoming TLS packet consists of stripping off the header during the read and performing a decrypt that moves the payload into place in the output stream. SSH in contrast requires a multi-stage processing operation that begins by decrypting the first 16 bytes of data (quantised to the cipher block size) and extracting the variable-length header to see how much more data needs to be processed. The non-header portions of the decrypted data are then MAC'ed and copied to the output. Finally, the remainder of the packet can be processed as normal. Because the header-processing forces a pipeline stall in the cryptologic, it's not possible to process an SSH packet using crypto hardware that requires a single-step operation.

A second, not directly performance-related problem with SSH is its handling of the message MAC. There are two approaches to encrypting and MACing messages, the first of which MACs the message and then encrypts it (MAC-then-encrypt) and the second of which encrypts the message and then MACs it (encrypt-then-MAC). Recent research regards encrypt-then-MAC as somewhat more secure than MAC-then-encrypt [49]. However, SSH does neither of these, first MACing the plaintext, then encrypting it, and finally appending the unencrypted MAC of the plaintext to the ciphertext. Even if it were modified to MAC the ciphertext, the fact that the header which indicates how much data needs to be MACed is encrypted still renders encrypt-then-MAC impossible. The design therefore manages to achieve none of the benefits of either encrypt-then-MAC or MAC-then-encrypt.

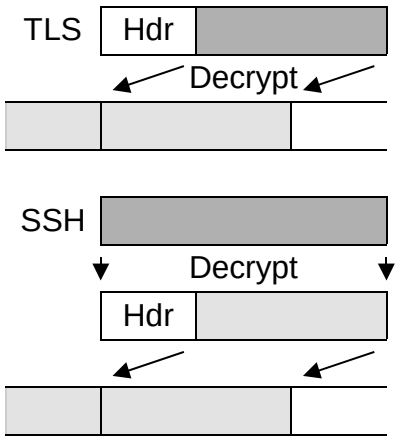

Figure 5: TLS (top) and SSH (bottom) message processing

The TLS 1.1 change to include an explicit IV in each message has little effect on processing provided that the hardware supports the use of a distinct IV as an input parameter (most does) or allows scatter/gather operation (compatibility of the explicit-IV feature with crypto hardware was explicitly taken into account during the TLS 1.1 design process). In the odd case where it doesn't, the IV can be treated as the -1^{th} encrypted data block, discarding it after decryption. Since this makes the combined decrypt-and-copy operation impossible, it's necessary to save the decrypted data already present as the first block, overwrite it with the decrypted output, and then replace the IV data in the -1^{th} block with its previously-saved contents (because of the variable-length data that precedes the payload, this same trick isn't possible with SSH's encrypted headers).

5.3 Encryption Hardware Performance

PKCS #11 is the standard interface for crypto hardware devices, supported by virtually all encryption hardware from simple smart cards and USB tokens through to high-end crypto accelerators and hardware security modules (HSMs). It has long been folklore in the PKCS #11 community that if your primary goal is high throughput, you only use the private-key acceleration capabilities of the hardware but do all other crypto on the host PC, because it's always faster to do symmetric crypto and hashing on the host. This is caused both by the straight hardware issues already discussed in section 5.1, and because of the overhead of moving each operation through one or more levels of drivers to get to the hardware. Given the AMD vs. Intel race for gigahertz in the last few years, even expensive public/private-key operations are frequently quicker in software on the host system than in hardware in the crypto device.

Providing absolute performance figures for PKCS #11 hardware is somewhat tricky because there's an awful lot of it around, with performance varying not only across devices but across different revisions of a device. For example one vendor implemented hashing using the device's controller CPU (a 25 MHz embedded 386 with a 16-bit bus) in an earlier version of their hardware, but used a hardware hash engine in later versions. Throughput jumped by more than two orders of magnitude in the newer devices. Other vendors have low-cost versions of devices (sold as general crypto accelerators) that only contain bignum (public/private-key) hardware, with more expensive all-round versions (sold as HSMs) containing symmetric crypto and hashing hardware as well. Still others perform hashing and some or all symmetric crypto on the host because it's so much faster than the crypto hardware.

Providing several pages of graphs of performance figures for various types of hardware isn't terribly meaningful, however a few observations about performance issues can be made for the pure hardware mechanisms (that is, when figures for crypto mechanisms implemented in software on the host are excluded). The most important observation is that the overhead for processing data quantities from 1 to 256 or 512 bytes (depending on the hardware, some are much worse than this) is more or less constant, indicating that the setup/driver overhead costs dominate the overall operation time. For the more lightweight hash algorithms, the breakeven point can go as high as 1-2 KB. This would indicate that for short data quantities, a considerable penalty is being paid for the access to external hardware. The exact impact of this depends on the application that's using the hardware. For example in a single SSL handshake IIS performs over six hundred (!!) separate hash operations, often on data quantities as small as one or

two bytes [50]. A better approach is to coalesce the data to be hashed to allow as much of a PDU to be processed at once as possible, which in cryptlib's case leads to a total of 65 hash operations per SSL handshake.

Even under optimal conditions though, the blocks never get very large. The largest possible SSL message size is 16 KB, with typical HTTP messages being in the 4-6 KB range [51], for SSH the maximum message size is application-specific but typically ranges from 4 to 32 KB, with 16 KB being a popular figure. SSH when used as a secure telnet replacement or for tunnelling data such as X Windows traffic can send large numbers of tiny messages, with a significant crypto call overhead for each message. The encryption (but not the hashing) overhead could in theory be reduced by using an encryption mode such as output feedback mode (OFB) that pre-generates keystream in bulk before it's needed and then applies it as required to short packets, however this rather obscure mode has virtually no support in actual encryption hardware.

For larger data blocks, the operation setup overhead no longer dominates, however even then all but the slowest CPUs outperform all but the fastest crypto hardware. Tests on a range of crypto hardware accelerators from various vendors provided a throughput for the worst-case algorithm for software, triple DES, of 5-15 MB/s depending on the hardware, roughly the same as a PC in the 1-2 GHz class [52]. For software-optimised algorithms such as SHA-1 and AES, the host CPU was always faster, even on the slowest system that the author had access to for hardware testing purposes, a 650 MHz Duron.

For devices not sold specifically as hardware accelerators (for example PCMCIA cards and USB tokens), the performance was predictable — all of them were far slower than the host. Many of these tokens perform symmetric crypto and hashing in software anyway, usually on ARM7-class hardware, for which Table 4 contains some typical timings.

These results would tend to confirm the folklore: unless the host is already at 100% capacity with other operations, it's better to perform symmetric encryption and hashing on the host rather than in dedicated hardware. The only real exception occurs when the encryption hardware is protocol-aware and can process complete security protocol PDUs at hardware speeds. For example the Broadcom device mentioned earlier can directly convert memory blocks to IPsec packets and back again, and devices from Cavium can do the same for SSL/TLS data via a built-in micro-sequencer that runs the TLS protocol directly on the crypto hardware [53]. Such special-purpose devices though are targeted at high-end networking appliances, and aren't found in general crypto accelerators.

5.4 Encryption Hardware as a Coprocessor

A question that arises from the above analysis is whether it's possible to derive some benefit from the fact that the crypto hardware can operate in parallel with the host CPU, performing its own operations while the host is busy with other tasks. For example since the host is obviously so much faster with hashing than the hardware, would it be possible to have the host hash the data while the hardware encrypts it? This also optimises use of memory bandwidth, since the CPU has to make a pass over the data anyway to hash it, it may as well pass it on directly to the crypto hardware while it's conveniently at hand. This optimisation, generally referred to as integrated layer processing (ILP), has been studied in some detail in the context of efficient network subsystem design [54]. The idea behind ILP is to load a block of data from memory once, move it through multiple manipulation layers, and then store the result in its destination location [55]. These combined encrypt-and-hash operations are used in SSH and TLS, as well as some variants of PGP. The general concept is shown in Figure 6 (top), with the CPU reading data from memory and feeding it on to the crypto hardware, which deposits it back in memory.

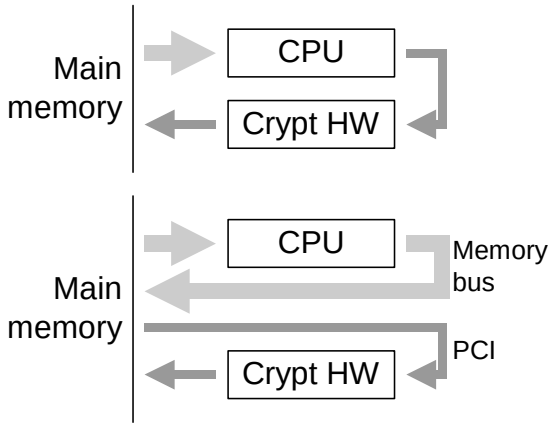

Figure 6: Planned (top) and actual (bottom) encryption hardware use

Unfortunately this isn't as simple as the diagram makes it appear. Once the data has passed across the memory bus through the CPU portion of the pipeline, it's necessary to use PIO to move it from there over the PCI bus to the hardware. An experimental implementation using a 3DES encryption card that supported a PIO interface produced results that were uniformly awful across several variations of transfer block size and synchronisation mechanisms used — the CPU spends more of its time taking care of communications housekeeping activities then doing anything else. As already mentioned in section 5.1, although the latency of an individual PIO transfer is lower than a DMA one, you can't beat a PCI burst read or write for throughput. A second problem is that this requires the use of a custom software interface, since PKCS #11 only supports memory-to-memory crypto operations and not (inherently non-portable) direct access to the encryption hardware.

A less optimal solution is shown in Figure 6 (bottom), with the CPU and crypto hardware processing the data in parallel. Unfortunately this alternative contains a dangerous race condition: If the hardware ever overtakes the CPU, the wrong data will be hashed. In general this will be unlikely since the high-speed CPU on a fast memory bus will always outperform the slower hardware on the PCI bus, however there's no guarantee that an interrupt or context switch won't stall the hashing long enough for the crypto to overtake it. Although the author wasn't able to create such a condition in his testing, it seems a considerable risk to take unless it's possible to do it in a custom driver that runs with interrupts disabled to ensure that the CPU always gets there first. In addition, the reverse situation occurs on decryption, with the CPU having to wait for the crypto hardware before it can hash the data. There isn't any easy way to synchronise the two operations, so they have to be performed in series.

Another possible performance optimisation is to combine the host-CPU hashing and coping of data to the caller's buffer/address space [56][57]. Unfortunately if a checksum failure occurs at this point the user will end up receiving invalid/corrupted data that then needs to be retroactively cleared when the problem is discovered, an unpleasant prospect for what's supposed to be a high-integrity security application. In addition, as with some of the other optimisations considered earlier, the combined hash+copy is a non-standard operation requiring custom modifications to existing software.

There is a third alternative possible with some crypto hardware that supports combined hash and encryption operations where both operations are performed in the hardware, taking advantage of parallelism inside the hardware rather than between host and hardware. This parallelism is supported in PKCS #11 via the `C_DigestEncryptUpdate` function, which first hashes the data being processed and then encrypts it. This means that the overall execution time of the operation will be the slower of the times of the individual encryption and hashing operations. In practice however the benchmarked times were all roughly the sum of both times, indicating that the two sets of operations were being performed in series rather than in parallel (these black-box results have since been confirmed by various hardware vendors). Since device (silicon die) size, cost, and power consumption/heat output considerations in a hermetically sealed HSM carry significant weight in these cases and the dual crypto operations are usually rarely used, it's likely that vendors chose to avoid the extra device complexity and did things the straightforward way (a number of vendors' drivers in fact perform the hashing and in some cases the symmetric crypto on the host system because it's so much faster than the hardware).

There are some high-performance crypto devices, generally reserved for use in high-end routers and similar special-purpose hardware rather than in general-purpose crypto cards, that do parallelise the encryption and hashing in the hardware. However these devices, targeted for IPsec use, have a different problem when used with application-level security protocols. IPsec performs its hashing after encryption (encrypt-then-hash) while the other protocols perform the encryption after hashing (hash-then-encrypt). This means that devices like the Broadcom 582x family mentioned earlier, which can perform encrypt-then-hash in a single operation, have to use two passes for hash-then-encrypt, with the host manually starting the second operation after the first one has completed. Unless your target is IPsec (for which performance is outstanding), you really can't win here.

6. Other Performance Issues

Beyond the protocol-specific issues discussed in the previous two sections, there are a number of other issues that affect security protocols, or that represent special-case protocol- or implementation-specific quirks that go beyond what's already been covered. This section looks at these additional issues, covering network performance issues and various protocol- and implementation-specific quirks.

6.1 Network Performance Issues

There are a variety of network performance issues that affect implementations of the secure session protocols. A simplistic implementation that performs two writes (the protocol header and the payload data written separately) followed by a read of the response will interact badly with TCP delayed-ACK and slow-start because the protocol header size is far less than the TCP MSS. The typical (Ethernet) LAN and WAN MTU is 1500 bytes for non-PPPoE implementations and 1492 bytes with the 8-byte PPPoE header, corresponding to an MSS of 1460 (non-PPPoE) or 1452 (PPPoE) bytes once the two 20-byte IP and TCP headers are taken into account. An older (and fortunately increasingly rare) MTU is 576 bytes, for an MSS of 536 bytes. A 576-byte MTU is also commonly used in dialup connections (for which performance is low anyway so it's not much of an issue), or a warning sign that PMTU discovery isn't enabled.

When the protocol handshake begins, the TCP congestion window starts at one segment, with the TCP slow-start then doubling its size for each ACK [58]. Sending the headers separately will send one short segment and a second MSS-size segment, whereupon the TCP stack will wait for the responder's ACK before continuing. The responder gets both segments, then delays its ACK for 200ms in the hopes of piggybacking it on response data, which is never sent since it's still waiting for the rest of the message body from the initiator. As a result, this results in a 200ms (+ assorted RTT) delay in each message sent. To avoid this problem, it's necessary to coalesce writes into a single atomic update that avoids (as far as possible) any interaction with TCP delayed-ACK and slow-start [59][60]. Another interaction with delayed-ACK, when no response data is available due to crypto processing overhead, has already been mentioned in section 4.2.

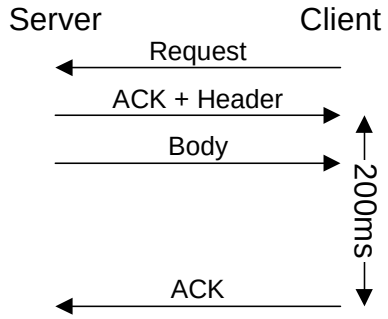

Figure 7: Delayed-ACK interaction with request/response protocol

The results of this interaction with TCP are shown in the packet trace in Figure 7. The extra overhead caused by the delayed ACK is clearly evident. When this problem was fixed in (at least) two LDAP servers, it resulted in a sixty-fold performance increase [61].

There is a somewhat related situation that occurs as a result of TCP slow-start in which it's not possible to send more than a single segment initially. This can't be avoided through clever programming, however most BSD-

derived implementations set the server's congestion window to two segments in response to receiving the TCP handshake ACK, so for the initial message exchange the client can send a request of 1MSS and the server a response of 2MSS without running into congestion-control problems.

A related problem is the fact that many TCP implementations will reset the congestion window after one retransmission timeout period if all data sent at that point has been ACK'ed, which means that both sides now restart with a congestion window of size of 1 or 2. This is because after a period of idle time the link characteristics may have changed, so that allowing an idle connection to resume with the same window it had when it left off would be equivalent to allowing a new connection to start with a large initial window, potentially flooding the link and causing packet loss. Unfortunately there's little that can be done about this apart from hoping that TCP implementations will start to fall into line with RFC 3390 and allow initial windows of ~4K, which will fix this particular problem. Other proposed alternatives include using longer restart timeouts, and slowly shrinking the window rather than reducing it in an all-or-nothing manner [62].

There are various other considerations that need to be taken into account in order to provide maximum efficiency. Fortunately these have been well-studied in the context of HTTP (which for non-persistent/streaming implementations run into these problems with every single transaction) and are covered in depth elsewhere [62][63][64]. In addition, modifications to TCP's behaviour such as the use of 4K initial windows (designed to reduce transfer times for small messages in protocols like HTTP to a single RTT) should also ameliorate some of these issues [65].

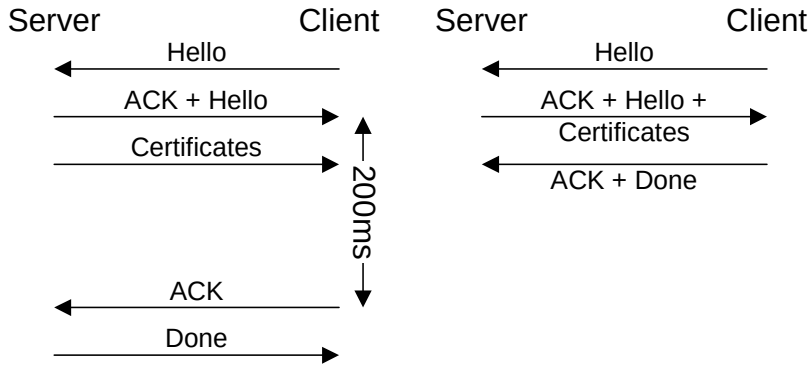

Figure 8: Delayed-ACK interaction with TLS/SSH exchange

The interaction of TLS and SSH with TCP are shown in Figure 8. On the left are simplistic implementations of TLS and SSH that write each packet as soon as it's ready, with TLS being used in the example (the process is identical for SSH). On the right is the corresponding optimised implementation, which bundles as many packets as possible before writing them, as per the ladder diagram in Figure 4. This task is somewhat easier in TLS than in SSH, since TLS bundles as many handshake packets as required into a single TLS record-layer packet, while SSH requires that each one be assembled as a discrete packet in a staging area before the whole collection is sent to the peer.

6.2 The SSHv2 and SFTP Performance Handbrake

In 1977, Ward Christensen created the Xmodem data transfer protocol [66]. Coming in an era of 300bps modems and unreliable links, Xmodem divided data into 128-byte packets and required an Ack to be sent for each packet before the next one could be transmitted, a so-called stop-and-wait protocol. Stop-and-wait protocols, although trivial to implement, are very inefficient, being warned against as far back as the 20+ year-old first edition of Computer Networks [67][68]. As modems became faster and links more reliable, the need to Ack each Xmodem packet became more and more of a performance handbrake, since no matter how fast or reliable the link, no more than 128 bytes of data could be sent without waiting 1 RTT for the Ack. The solution to the problem was to increase the packet size (Ymodem, Xmodem-1K), and drop the requirement to Ack each packet (Ymodem-g, Zmodem) [69]. The latter was perfectly acceptable, since by then modems included their own error correction and flow control mechanism.

Unfortunately this performance handbrake was reinvented in the SSHv2 protocol. Like Ymodem-g and Zmodem running over modern modems, TCP/IP provides a reliable, flow-controlled transport layer for the SSH protocol.

SSHv2 however introduced an additional form of flow control that, like Xmodem, requires the receiver to Ack each packet before more can be sent (the details aren't quite as straightforward as this since the SSHv2 specification describes things in terms of packets and data windows, but effectively it's the Xmodem per-packet Ack). Most implementations seem to use packet sizes of 16K or occasionally 32K, with some going as low as 4K. What this means is that no matter how fast the link, every (say) 16K the transmission stops for 1 RTT until the other side has sent its Ack (referred to as a window adjust in SSHv2 terminology). Consider for example the effect of this on a T1 international link with a half-second RTT. With the handbrake in operation, the link can run at only 17% of its total capacity. This performance hit is so noticeable that it's mentioned in the FAQs of some SSH implementations [70], is a frequent topic of discussion in fora such as the SSH usenet newsgroup, and has been demonstrated by other researchers [1]. The general problem of tunnelling TCP over TCP has been pointed out by others [71].

In addition to the protocol-level handbrake, the SFTP protocol that runs on top of SSH contains its own handbrake. This protocol recommends that reads and writes consist of no more than 32K of data, even though it's running over the reliable SSH transport which is in turn running over the reliable TCP/IP transport. One common implementation limits SFTP packets to 4K bytes, resulting in a mere 4% link utilisation in the previously-presented scenario.

There are two possible solutions to this problem, either to try to ameliorate the effects of the handbrake or to remove it altogether. Unfortunately trying to ameliorate this problem isn't really practical. While it's possible to carefully tune stop-and-wait protocols to perform less poorly under the right circumstances, this requires a considerable amount of work, the right operating conditions, and two communicating implementations that are tuned it the same way. For example the Kermit protocol has an almost universal reputation for poor performance because, although a tuned software implementation using sliding windows exists, the large number of other implementations were never aware of the problem, and even if they were it was easier to switch to a protocol without the problem than to try and get stop-and-wait to perform effectively. This was why Kermit was almost completely displaced by protocols like Zmodem in its field of application.

The same situation is being repeated with SSH, with large numbers of independent implementations by vendors that aren't aware of the problem (the one published design document of a commercial SSH product contains no mention of the handbrake [72], and its presence has been verified in a number of other implementations). Even if vendors were aware of it, they are unlikely to be motivated to fix it, since many commercial vendors have implemented SSH as an add-on to an existing product and barely fix obvious implementation bugs, let alone taking the time to experiment with complex protocol tuning, particularly when their implementation "works" as is.

The simpler solution is to remove it entirely, since when performing a straightforward data transfer (without multiplexing multiple channels over the same link), there is no good reason for the per-packet Ack, and certainly other protocols such as SSHv1 and SSL/TLS function perfectly without it (the absence of the handbrake in SSHv1 is why SSH FAQs observe that the SSHv1 scp is so much faster than the SSHv2 SFTP, even though SFTP is overall a better design). The effect of running without the handbrake on were investigated using cryptlib with a fairly rudimentary implementation of SFTP running over the built-in SSHv2. cryptlib's SSH implementation has always set the window size to INT_MAX (some implementations have problems with UINT_MAX as the window size), which effectively disables the SSH-level handbrake. The SFTP implementation followed suit, requesting a read/write of the entire file at once rather than breaking it up into little packets at the SFTP level (packetisation is already handled at the SSH and TCP/IP layers). Run over an international link (pretty much a given when you're in New Zealand), this SFTP implementation was around five times faster than the Putty implementation of SFTP talking to OpenSSH, which sends data in 4K SFTP packets and (by extension) 4K SSH packets. Even over a low-latency link, the difference was impressive: cryptlib was an order of magnitude faster than Putty on the loopback interface (latency being relative in this case). An experimental add-on to OpenSSH, HPN-SSH, found a similar order-of-magnitude performance increase over an unmodified implementation [73],however this add-on version requires that both sides be using the HPN implementation in order to see any improvement.

The SSH-level handbrake can therefore be provisionally removed by having implementations set the window size to INT_MAX, and permanently removed by deprecating the Ack/window-based flow control. Instead of the existing Xmodem-style deny-until-told-otherwise flow control mechanism, a new Zmodem-style allow-until-told-otherwise mechanism that implements Xon/Xoff-style flow control can be added for use where appropriate (as was mentioned earlier, both SSHv1 and SSL/TLS function perfectly without requiring this). The SFTP-level handbrake can be removed by eliminating the maximum packet-size wording of the SFTP specification, and recommending that implementations read and write all data at once rather than engaging in additional redundant packetisation at the SFTP level.

There exist situations in which the complete absence of flow control can be a problem. In cases where SSH is being used as a secure FTP/rcp, in which the transfer model is to open the connection and push data across it as quickly as possible until either the transfer completes or an error causing it to fail occurs, there's no need for further flow control (FTP and rcp work just fine without it). In these cases the simple infinite window-size workaround suffices. However, where SSH is being used as a VPN with multiple transfers active at once (in SSH terms there are multiple channels active), it's necessary to provide per-channel flow control to handle situations with slow consumers on individual channels [74].

Most of the necessary changes to avoid the handbrake can be effected through a simple code update, however implementers should be aware that some implementations will still stop and wait for an Ack after a certain amount of data has been transmitted, even with effectively infinite-sized windows. On the sender side things aren't quite so bad, experimentation has shown that it's fairly safe to ignore the receiver's window size and send data at the maximum rate possible, discarding any window adjusts that arrive (the only slight complication is that it's occasionally necessary to stop sending for a moment and clear the read channel of the accumulation of Acks that have arrived while sending). Since most implementations include a facility for checking the peer's software version to identify and work around implementation bugs, detecting pre-handbrake-fix implementations and providing the appropriate slower interpretation of the protocol should be relatively straightforward. In addition, FAQs about the poor performance of SFTP will need to be updated.

6.3 Other Issues

The process of instrumenting the code and obtaining performance figures yielded some surprising results regarding the performance of the bignum code. This was notoriously malloc-intensive up until a few years ago and had been benchmarked at the time to spend up to 10% of its overall runtime inside malloc, but an attempt had been made at that time to try and address this. However, even the very latest version of the code produced around a page of malloc calls when performing a key load (like many other libraries, cryptlib performs a number of key-validation computations on a key when it loads it), and a second page of malloc calls when performing a sample signature-generation operation. Apart from the obvious overhead of spending so much time inside malloc, this has performance implications on multithreaded systems and serious performance implications on multiprocessor ones. On any multithreaded system, access to the heap must be serialised using a mutex, with every malloc potentially resulting in a context switch if several threads are active. On multiprocessor systems it's even worse since, once all threads scheduled on it are waiting on the malloc mutex, the CPU is effectively disabled.

By wrapping the memory allocation calls in a fast (external) mutex and running the crypto operations in a loop on an older dual-processor system, it was possible to observe a regular pileup of threads fighting over heap access (the Win32 heap has historically exhibited poor performance on multiprocessor systems [75][76]). This conflict mostly disappeared on a relatively new dual-processor system running Windows Server 2003, although it wasn't possible to determine whether this was due to an improved memory management subsystem or because of some other aspect of the OS or hardware (there are documents claiming enhancements to the heap management to improve scalability for this version of the OS [77], but there are documents with similar claims going back to at least Windows NT 4.0). Although the tests were run on Windows machines, other multiprocessor systems have similar problems [78].

In order to determine the best strategy for resolving these memory problems, we need to examine how a modern high-performance memory allocator works [79]. In order to increase performance, these typically contain multiple free lists for small fixed-size blocks (usually sized in powers of two starting from 8 or 16 bytes), and a general grab-bag list for larger blocks. Depending on the strategy used to handle the fixed-size block list, a request for a small non power-of-two block will either return a block of the nearest power-of-two size, or be sent to the grab-bag list. In both cases rounding up the allocation to the nearest power of two will provide some overflow space for the bignum while having no impact on performance and little or no impact on memory use.

Update with more info on dynAlloc'd version, which overallocates initially and then uses a high-water-mark allocator Insert traces of what reallocs do – usually we double the bignum size, so need to overestimate to nearest power-of-two x 2.

7. Conclusion

This paper has presented an analysis of the four most popular application-level security protocols, PGP and S/MIME for fixed message protection and SSL/TLS and SSH for communications session protection. When the message

format is chosen to minimise message size, there is very little difference either in overall size or processing overhead between PGP and S/MIME. Unfortunately the default message format used for S/MIME messages tends to blow the size out due to the use of X.509 certificates and certificate-related data structures.

Compare to TLS, SSH loses out due to its protracted handshake negotiations and the need to apply multiple heavyweight public/private-key operations against TLS' one (or zero for TLS-sharekey). This interacts badly with network characteristics such as TCP delayed-ACK, and leads to poor performance on low-powered devices.

Beyond these general results, a number of recommendations for designers, implementers, and users of the various protocols has emerged:

- When designing a new protocol, make sure that the sections subject to crypto processing are nicely aligned on 64-bit boundaries to allow optimal handling by crypto hardware.
- Don't create a message format that requires complex two-stage processing, for example by encrypting the message headers.
- Always benchmark your code to identify anomalous behaviour. Don't assume that just because a particular algorithm or implementation is supposed to exhibit certain characteristics, that it actually does [80].
- Remember that not everyone is working with the latest, fastest CPU hardware with (effectively) infinite amounts of memory. A protocol design may work fine on a 3 GHz CPU, but how well does it work on one that's two orders of magnitude slower and has 64KB RAM? The protocol should at least provide an alternative mechanism for use on resource-constrained systems, such as TLS' shared-key mechanism, that doesn't require public-key crypto operations or certificates.
- Conversely, if you have a fast enough host CPU, it isn't worth using dedicated crypto hardware for symmetric crypto and hashing unless the CPU is already at maximum utilisation anyway.

And one final conclusion, which has probably become obvious to anyone reading this far:

- Providing any kind of one-size-fits-all performance figure (or set of figures) for the various application-level security protocols isn't possible because of a large number of situation-specific variations that can affect performance. It's necessary to examine the internals of each protocol in some detail in order to obtain a true indication of performance issues and potential problem areas. Conversely, it's possible to obtain considerable performance improvements through relatively simple implementation options like choice of algorithm and caching keys instead of re-importing them on each use.

8. Acknowledgements

The author would like to thank Alan Braggins, Matthis Bruestle, Jon Callas, Joan Dyer, Laszlo Elteto, Martin Forssen, Perry Metzger, Colin Plumb, Blake Ramsdell, Eric Rescorla, and Phil Zimmermann for advice on protocol-specific and hardware issues and help with obtaining timings on embedded systems.

9. References

[1] "A Study of the Relative Costs of Network Security Protocols", Stefan Miltchev, Sotiris Ioannidis, and Angelos Keromytis, *Proceedings of the 2002 Usenix Annual Technical Conference (Freenix track)*, June 2002, p.41.

[2] "Performance Analysis of TLS Web Servers", Cristian Coarfa, Peter Druschel, and Dan Wallach, *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'02)*, February 2002, p.183.

[3] "Remote Timing Attacks Are Practical", David Brumley and Dan Boneh, *Proceedings of the 2003 Usenix Security Symposium (Security'03)*, August 2003, p.1.

[4] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 1999 Usenix Security Symposium (Security'99)*, August 1999, p.153.

[5] "Cryptographic Security Architecture: Design and Verification", Peter Gutmann, Springer-Verlag, 2003.

[6] "Digital Signature Standard (DSS)", FIPS PUB 186-2, National Institute of Standards and Technology, February 2000.

[7] "Security Requirements for Cryptographic Modules", FIPS PUB 140-2, National Institute of Standards and Technology, July 2001.

[8] `/dev/random` driver source code (random.c), Theodore T'so, 24 April 1996.

[9] "HOWTO: Export/Import Plain Text Session Key Using CryptoAPI", Microsoft Knowledge Base Article 228786.

[10] "OpenPGP Message Format", RFC 2440, Jon Callas, Lutz Donnerhacke, Hal Finney, Rodney Thayer, November 1998.

[11] "OpenPGP Message Format", IETF draft, Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer, 2003.

[12] "PKCS #7: Cryptographic Message Syntax, Version 1.5", RSA Data Security, Inc, 1993.

[13] "S/MIME Version 2 Message Specification", RFC 2311, Steve Dusse, Paul Hoffman, Blake Ramsdell, Laurence Lundblade, and Lisa Repka, March 1998.

[14] "S/MIME Version 2 Certificate Handling", RFC 2312, Steve Dusse, Paul Hoffman, Blake Ramsdell, and Jeff Weinstein, March 1998.

[15] "Cryptographic Message Syntax (CMS)", RFC 3369, Russ Housley, August 2002.

[16] "Cryptographic Message Syntax (CMS) Algorithms", RFC 3370, Russ Housley, August 2002.

[17] "S/MIME Version 3 Message Specification", RFC 2632, Blake Ramsdell, June 1999.

[18] "S/MIME Version 3 Certificate Handling", RFC 2632, Blake Ramsdell, June 1999.

[19] "The SSL Protocol, Version 3.0", Alan Freier, Philip Karlton, and Paul Kocher, Netscape Communications, 18 November 1996.

[20] "The TLS Protocol, Version 1.0", RFC 2246, Tim Dierks and Christopher Allen, January 1999.

[21] "The TLS Protocol, Version 1.1", IETF draft, Tim Dierks and Eric Rescorla, 2003.

[22] "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", Bodo Moeller, 2002, `http://www.openssl.org/~bodo/tls-cbc.txt`.

[23] "Attacking Predictable IPsec ESP Initialisation Vectors", Sami Vaarala, Anti Nuopponen, and Teemupekka Virtanen, *Proceedings of the 4th International Conference on Information and Communications Security (ICICS'02)*, Springer-Verlag LNCS No.2513, December 2002, p.160.

[24] "SSH Protocol Architecture", IETF draft, Tatu Ylonen and Darren Moffat, 2003.

[25] "SSH Authentication Protocol", IETF draft, Tatu Ylonen and Darren Moffat, 2003.

[26] "SSH Connection Protocol", IETF draft, Tatu Ylonen and Darren Moffat, 2003.

[27] "SSH Transport Layer Protocol", IETF draft, Tatu Ylonen and Darren Moffat, 2003.

[28] "Virtual Private(?) Networks", Peter Gutmann, *Linux Magazine*, **Issue 39** (February 2004), p.*to appear*.

[29] "PGP Message Exchange Formats", RFC 1991, Derek Atkins, William Stallings, and Philip Zimmermann, August 1996.

[30] "Format of a Signed File", Peter Gutmann, 1997, `http://www.cs.auckland.ac.nz/~pgut001/pubs/authenticode.txt`.

[31] "Authenticode Overviews and Tutorials", Microsoft Developer Network (MSDN), 2002.

[32] "algorithm ID encodings & terminology", ietf-open-pgp mailing list thread started by Adam Back, 8 August 1997.

[33] "hand huffman encoding at PGP world HQ", ietf-open-pgp mailing list thread started by Adam Back, 23 November 1997.

[34] Private communication with PGP developers.

[35] "AKI and SKI problem with RFC 3280?", ietf-pkix mailing list thread started by Stefan Santesson, 20 October 2003.

[36] "Playing Hide and Seek with Stored Keys", Adi Shamir and Nicko van Someren, *Proceedings if the 3rd International Conference on Financial Cryptography (FC'99)*, Springer-Verlag LNCS No.1648, February 1999, p.118.

[37] "Diffie-Hellman Group Exchange for the SSH Transport Layer Protocol", IETF draft, Markus Friedl and William Simpson, 2003.

[38] "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem", Jean-Jacque Quisquater and C.Couvreur, *Electronics Letters*, **Vol.18**, **No.21** (October 1982), p.905.

[39] "Handbook of Applied Cryptography", Alfred Menezes, Paul van Oorschot, and Scott Vanstone, CRC Press, October 1996.

[40] "OpenCores.ORG — Microprocessors", `http://www.opencores.org/projects?category=microprocessor`.

[41] "Use of Shared Keys in the TLS Protocol", Peter Gutmann, IETF draft, October 2003.

[42] "PCI Local Bus Specification, Revision 2.1", PCI Special Interest Group, June 1995.

[43] "PCI System Architecture (4th ed)", Tom Shanley and Don Anderson, Addison-Wesley, 1999.

[44] "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", Peter Druschel and Larry Peterson, *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, December 1993, p.189.

[45] "An Efficient Zero-Copy I/O Framework for UNIX", Moti Thadani and Yousef Khalidi, Sun Microsystems Lab Technical Report SMLI TR-95-39, May 1995.

[46] "Operating System Support for High-Speed Communication", Peter Druschel, *Communications of the ACM*, **Vol.39**, **No.9** (September 1996), p.41

[47] "Operating System Support for High-Performance Networking: A Survey", Peng Wang and Zhiqing Liu, 2001, `http://www.cs.iupui.edu/~zliu/doc/os_survey.pdf`.

[48] "Real-time Certificate Status Facility for CMS - (RTCS)", IETF draft, Peter Gutmann, June 2003.

[49] "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", Hugo Krawczyk, *Advances in Cryptology (Crypto'01)*, Springer-Verlag LNCS No.2139, August 2001, p.310.

[50] "Re: Highly optimized crypto APIs", Laszlo Elteto, posting to the pkcs-tng@rsa.com mailing list, message-ID `5606C687D4C7D5119A5800508BF30DB40154BD68@mail.rainbow.com`, 12 April 2002.

[51] "Summary of WWW Characterizations", James Pitkow, *Proceedings of the 7th International World Wide Web Conference / Computer Networks*, **Vol.30**, **No.1-7** (1 April 1998), p.551.

[52] "Fast Implementations on the Pentium", Antoon Bosselaers, `http://www.esat.kuleuven.ac.be/~bosselae/fast.html`.

[53] "Cavium Networks Products", `http://www.cavium.com/processors.htm`.

[54] "Network Subsystem Design: A Case for an Integrated Data Path", Peter Druschel, Mark Abbott, Michael Pagels, and Larry Peterson, *IEEE Network*, **Vol.7**, **No.4** (July 1993), p.8.

[55] "Increasing network throughput by integrating protocol layers", Mark Abbott and Larry Peterson, *IEEE/ACM Transactions on Networking*, **Vol.1**, **No.5** (October 1993), p.600.

[56] "Modularity and Efficiency in Protocol Implementation", RFC 817, David Clark, July 1982.

[57] "Architecture considerations for a new generation of protocols", David Clark and David Tennenhouse, *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'90)*, September 1990, p.200.

[58] "Congestion Avoidance and Control", Van Jacobson, and Michael Karels, *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'88)*, August 1988, p.314.

[59] "Application performance pitfalls and TCP's Nagle algorithm", Greg Minshall, Yasushi Saito, Jeffrey Mogul, and Ben Verghese, *ACM SIGMETRICS Performance Evaluation Review*, **Vol.27**, **No.4** (March 2000), p.36.

[60] "Re: TCP/IP: Delayed ACK, Push, NODELAY & Nagle", John Nagle, posting to comp.protocols.tcp-ip newsgroup, 25 April 1997, message-ID `nagleE96EyD.Gx2@netcom.com`.

[61] "Re: Setting TCP_NODELAY on TCP sockets", David Boreham, posting to the openldap-devel@OpenLDAP.org mailing list, 22 July 1999, message-ID 37975ACB.BDA6CBA8@netscape.com.

[62] "Performance Interactions Between P-HTTP and TCP Implementations", John Heidemann, *ACM Computer Communications Review*, **Vol.27**, **No.2** (April 1997), p.65.

[63] "Analysis of HTTP Performance Problems", Simon Spero, July 1994, http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html.

[64] "Network Performance Effects of HTTP/1.1, CSS1, and PNG", Henrik Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley, 24 June 1997, http://www.w3.org/Protocols/HTTP/1.0/Performance/Pipeline.html.

[65] "Increasing TCP's Initial Window", RFC 3390, Mark Allman, Sally Floyd, and Craig Partridge, October 2002.

[66] "MODEM.ASM", Ward Christensen, August 1977 (the Xmodem protocol was defined in terms of "What this program does" rather than being formally documented, the author described it in a Compuserve post some years later as "a quick hack I threw together").

[67] "Computer Networks", Andrew Tanenbaum, Prentice-Hall, 1981.

[68] "Network Protocols", Andrew Tanenbaum, *ACM Computing Surveys*, **Vol.13**, **No.4** (December 1981), p.453.

[69] "Xmodem/Ymodem Protocol Reference: A compendium of documents describing the Xmodem and Ymodem File Transfer Protocols", Chuck Forsberg, October 1988.

[70] "PuTTY FAQ", Simon Tatham, 2003, http://www.chiark.greenend.org.uk/~sgtatham/putty/faq.html, question A.6.8, "PSFTP transfers files much slower than PSCP".

[71] "Why TCP over TCP is a bad idea", Olaf Titz, http://sites.inka.de/bigred/devel/-tcp-tcp.html.

[72] "The Secure Shell Game", Glen Mathews, *Dr.Dobbs Journal*, **Vol.29**, **No.6** (June 2004), p.38.

[73] "High Performance Enabled SSH", Chris Rapier, http:///www.psc.edu/networking/-projects/hpn-ssh/.

[74] "Re: Why SFTP performance sucks, and how to fix it", Markus Friedl, posting to the ietf-ssh@netbsd.org mailing list, message-ID 200307080401.GA14517@folly, 8 July 2003.

[75] "Heap: Pleasures and Pains", Murali Krishnan, MSDN Library (CDROM edition), February 1999.

[76] "Server Performance and Scalability Killers", George Reilly, MSDN Library (CDROM edition), 22 February 1999.

[77] "Windows Server 2003 Kernel Scaling Improvements", Microsoft Corporation, 6 October 2003.

[78] "Hoard: A Scalable Memory Allocator for Multithreaded Applications", by Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson, *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. November 2000, p.117.

[79] "Dynamic Storage Allocation: A Survey and Critical Review", Paul Wilson, Mark Johnstone, Michael Neely, and David Boles, *Proceedings of the International Workshop on Memory Management*, September 1995.

[80] "Learning security QA from the vulnerability researchers", Chris Wysopal, *;login*, **Vol.28**, **No.6** (December 2003), p.13.