The Impact of TLS on SIP Server Performance

Charles Shen[†] Erich Nahum[‡] Henning Schulzrinne[†] Charles Wright[‡]

[†]Department of Computer Science, Columbia University
New York, NY 10027, USA
{charles,hgs}@cs.columbia.edu

[‡]IBM T.J. Watson Research Center
Hawthorne, NY 10532, USA
{nahum,cpwright}@us.ibm.com

ABSTRACT

Securing VoIP is a crucial requirement for its successful adoption. A key component of this is securing the signaling path, which is performed by SIP. Securing SIP is accomplished by using TLS instead of UDP as the transport protocol. However, using TLS for SIP is not yet widespread, perhaps due to concerns about the performance overhead. This paper studies the performance impact of using TLS as a transport protocol for SIP servers. We evaluate the cost of TLS experimentally using a testbed with OpenSIPS, OpenSSL, and Linux running on an Intel-based server. We analyze TLS costs using application, library, and kernel profiling, and use the profiles to illustrate when and how different costs are incurred, such as bulk data encryption, public key encryption, private key decryption, and MAC-based verification. We show that using TLS can reduce performance by up to a factor of 17 compared to the typical case of SIP-over-UDP. The primary factor in determining performance is whether and how TLS connection establishment is performed, due to the heavy costs of RSA operations used for session negotiation. This depends both on how the SIP proxy is deployed (e.g., as an inbound or outbound proxy) and what TLS options are used (e.g., mutual authentication, session reuse). The cost of symmetric key operations such as AES, in contrast, tends to be small.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Communications Applications; B.8 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement, Experimentation, Security

IPTComm 2010, 2-3 August, 2010 Munich, Germany

Keywords

SIP, TLS, Security, Performance

1. INTRODUCTION

Securing Voice over IP (VoIP) is a necessary requirement for enabling its stable, long-term adoption. A key aspect of VoIP security is securing the signalling path, typically provided by the Session Initiation Protocol (SIP) [35]. SIP is an application layer signaling protocol for creating, modifying, and terminating media sessions in the Internet. Major standards bodies including 3GPP, ITU-T, and ETSI have all adopted SIP as the core signaling protocol for services such as VoIP, conferencing, Video on Demand (VoD), presence, and Instant Messaging (IM). Like other Internet services, SIP-based services may be susceptible to a wide variety of security threats including social threats, traffic attacks, denial of services and service abuse [3, 7, 22]. One of the main reasons that enable these threats is the common use of insecure SIP signaling such as SIP-over-UDP, which provides no signaling confidentiality, integrity, or authenticity. Given a trace of SIP traffic, one can see who is communicating with whom, when, for how long, and sometimes even what is being said (e.g., in SIMPLE [8]). It has also been shown that even commercial VoIP services may be prone to large-scale voice pharming [41], where victims are directed to fake interactive voice response systems or human representatives for revealing sensitive information.

Transport Layer Security (TLS) [15] is a widely used Internet security protocol occupying a layer between the application and a reliable transport like TCP. There is also a Datagram TLS (DTLS) [33] protocol that provides similar security functionalities but runs over an unreliable transport like UDP. The SIP specification [35] lists TLS as a standard method to secure SIP signaling. Various other organizations and industrial consortiums have also recommended or mandated the use of TLS for SIP signaling. For example, the SIP Forum [2] mandated TLS for interconnecting enterprise and service provider SIP networks in its specification document.

However, while interest in securing SIP is growing [31], actual large scale deployment of SIP-over-TLS has not yet occurred. One important reason is the common perception that running an application over TLS is costly compared to running it directly over TCP (or UDP in the case of SIP). VoIP providers will be hesitant to deploy TLS until they understand the resource provisioning and capacity planning

required. Thus we need to understand how much using TLS with SIP actually costs.

This paper makes the following contributions:

- We present an experimental performance study of the impact of using TLS on SIP servers. Our study is conducted using OpenSIPS [27] with OpenSSL [28] on Linux on an Intel-based server. We evaluate the CPU cost of TLS under four SIP proxy usage scenarios: proxy chain, outbound proxy, inbound proxy, and local proxy. We show that using TLS can reduce performance by up to a factor of 17 compared to the typical case of SIP-over-UDP.

- We use application, library, and kernel profiles to examine, analyze, and explain performance differences. The profiles illustrate how costs are incurred under different scenarios (e.g., extra Rivest, Shamir and Adleman (RSA) overheads when mutual authentication is used) and how the costs can be reduced (e.g., RSA costs disappear when session reuse is performed). They also show some results that distinguish SIP server from other server scenarios (e.g., bulk crypto costs of Advanced Encryption Standard (AES) or Triple Data Encryption Standard (3DES) are small) and how some overheads are due to mechanisms (e.g., kernel overhead, Secure Sockets Layer (SSL) state management) rather than simply crypto algorithms such as RSA or AES.

- We identify and solve two performance problems in OpenSIPS. Each is related to connection management with large numbers of connections under high loads. The fixes improve performance in some cases from a few times up to an order of magnitude.

Previous studies on TLS performance have either focused on TLS for Web servers [5, 10, 21, 44] or policy-based network management [43]. SIP protocol behavior is different from these protocols in several ways. SIP messages tend to be small, whereas Web downloads can be quite large. SIP proxy servers can act as clients to other servers and therefore can incur large client-side TLS costs. Finally, SIP servers have a much wider range of connection management behavior than other servers, and this connection management is the primary issue in determining TLS overheads, due to the heavy costs of RSA operations used for session negotiation. Symmetric key operations such as AES or 3DES are trivial in comparison.

The net result is that the performance cost of deploying SIP over TLS instead of UDP can be significant, depending on how the SIP proxy server is deployed (e.g., as an inbound or outbound proxy) and how TLS is configured (e.g., with or without mutual authentication or session reuse). Network operators can minimize this cost by attempting to maximize the persistence of secure sessions, but still need to be aware of the overhead of utilizing TLS.

The remainder of this paper is structured as follows. Section 2 provides some background on TLS and SIP. Section 3 describes the testbed used and how we determine our test cases. Section 4 presents our experimental results. Section 5 describes related work.

2. BACKGROUND

2.1 TLS Operation Overview

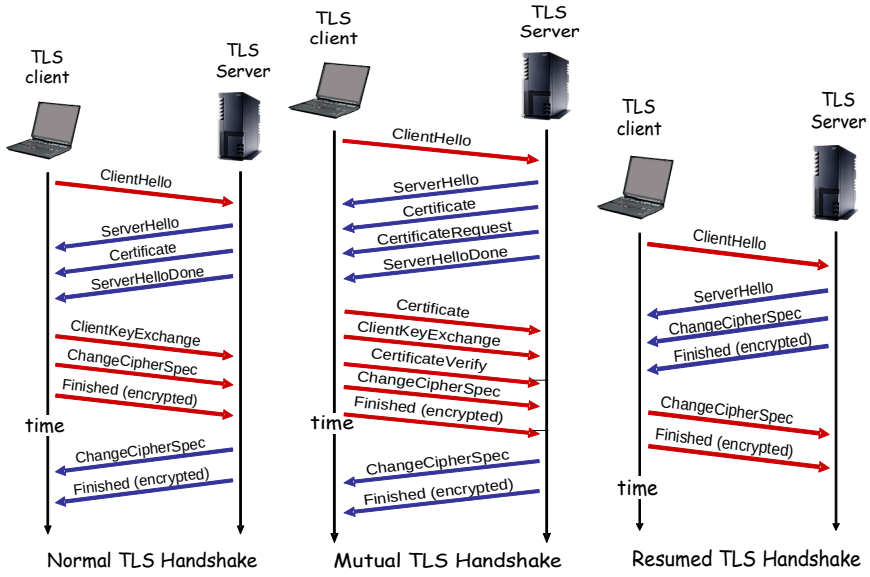
We provide a brief description of the TLS protocol. For more details, please see [15, 32, 37]. We focus on the aspects relevant to our study, namely session establishment and its corresponding RSA public key costs.

TLS operation consists of two phases: the handshake phase and the bulk data encryption phase. The handshake phase allows the parties to negotiate the algorithms to be used during this TLS session, authenticate the other party and prepare the shared secret for the bulk data encryption phase.

The normal message flow in the TLS handshake phase is illustrated in Figure 1(a). The key performance aspects of this handshake are that the server sends its certificate to the client, which then verifies the certificate using a certificate authority's public key. Depending on the key exchange mode, the client may then generate a `pre_master_secret`, and encrypt it using the server's public key obtained from the server's certificate. The server decrypts the `pre_master_secret` using its own private key. Both the server and client then compute a `master_secret` they share based on the same `pre_master_secret`. The `master_secret` is further used to generate the shared symmetric keys for bulk data encryption and message authentication.

In normal TLS handshake, only the client authenticates the server. In situations where the server also wishes to authenticate the client, TLS provides a mutual authentication mode as shown in Figure 1(b). In the mutual authentication mode, after the server sends its own certificate to the client, the server sends an additional `CertificateRequest` message to request the client's certificate. The client responds with two additional messages, a `Certificate` message containing the client certificate with the client public key, and a `CertificateVerify` message containing a digest signature of the handshake messages signed by the client's private key. Since only a client holding the correct private key can sign the message, the server can authenticate the client using the client's public key.

In general, public key cryptographic operations such as RSA are much more expensive than shared key cryptography. This is why TLS uses public key cryptography to establish the shared secret key in the handshake phase, and then uses symmetric key cryptography with the negotiated shared secret as the data encryption key. TLS offers a session reuse mode where the two parties can avoid negotiating the `pre_master_secret` if it has been done previously within some time threshold. It is important to distinguish the notion of a *connection* versus a *session* in TLS. A TLS *connection* corresponds to one specific communication channel which is typically a TCP connection; while a TLS *session* is associated with a negotiated set of algorithms and the established `master_secret` based on the `pre_master_secret`. Multiple connections may be mapped to the same session, all sharing the same set of algorithms and the `master_secret`, but each with different symmetric keys for bulk data encryption. The notion of session reuse indicates the reuse of a previously negotiated set of cryptographic algorithms and the `master_secret`. The handshake message flow for TLS session reuse is shown in Figure 1(c). The first time the client and server communicate, they establish a new connection and a new session. The server stores the session information including the algorithm choice and the `master_secret` for later reference. The session is identified by a `session_id` which is conveyed to the client during the initial



(a) normal (b) with mutual authentication (c) with session reuse

Figure 1: TLS Handshake Message Flows

handshake in the `ServerHello` message. The next time the client needs to establish a connection, it can include the previous `session_id` in the `ClientHello` message. The server agrees to session reuse by specifying the same `session_id` in the responding `ServerHello` message. The TLS handshake will then proceed to `ChangeCipherSpec` message and `Finished` message directly, avoiding the re-computation of a `pre_master_secret`. The session reuse timeout is configurable based on the security assumptions of how long it takes to break the key by brute-force.

2.2 SIP Overview

SIP defines two basic types of entities: User Agents (UAs) and servers. UAs represent SIP end points. SIP servers consist of registrar servers for location management, and proxy servers for message forwarding. SIP messages are divided into requests (e.g., `INVITE` and `BYE` to create and terminate a SIP session, respectively) and responses (e.g., `200 OK` for confirming a session setup). The set of messages including a request and its associated responses is called a SIP transaction.

SIP message forwarding, known as proxying, is a critical function of the SIP infrastructure. This forwarding process is provided by proxy servers and can be either stateless or stateful. Stateless proxy servers do not maintain state information about the SIP session and therefore tend to be more scalable. However, many standard application functionalities, such as authentication, authorization, accounting, and call forking require the proxy server to operate in a stateful mode by keeping different levels of session state information. Therefore, we focus on stateful SIP proxying in this paper.

Figure 2 shows a typical message flow of stateful SIP proxying with authentication enabled. Two SIP UAs, designated as User Agent Client (UAC) and User Agent Server (UAS) represent the caller and callee of a multimedia session. The hashed circle around the proxy indicates that this is the

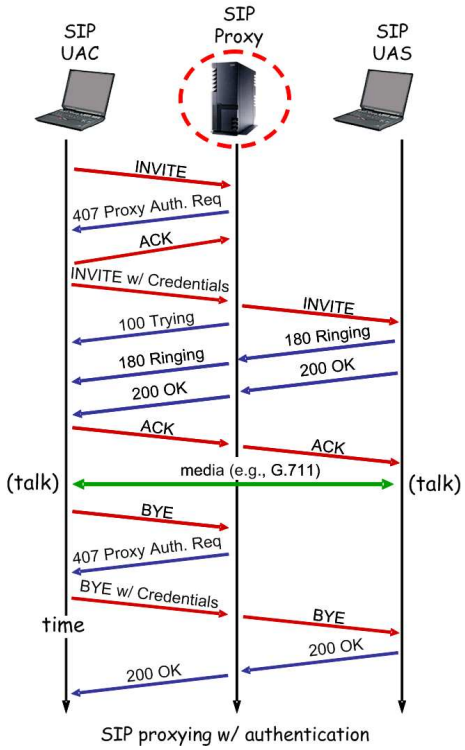


Figure 2: SIP Stateful Proxying with Authentication

server that we are measuring ("system under test"). In this example, the UAC wishes to establish a session with the UAS and sends an `INVITE` message to the proxy. The proxy server enforces proxy authentication and responds with a `407 Proxy Authentication Required` message, challenging the UAC to provide credentials that verify its claimed identity (e.g., based on MD5 [34] digest algorithm). The UAC then retransmits the `INVITE` message with the generated credentials in the `Authorization` header. After receiving and verifying the UAC credential, the proxy sends a `100 TRYING` message to inform the UAC that the message has been received and that it needs not worry about hop-by-hop retransmissions. The proxy then looks up the contact address for the SIP URI of the UAS and, assuming it is available, forwards the message. The UAS, in turn, acknowledges receipt of the `INVITE` message with a `180 RINGING` message and rings the callee's phone. When the callee actually picks up the phone, the UAS sends out a `200 OK`. Both the `180 RINGING` and `200 OK` messages make their way back to the UAC through the proxy. The UAC then generates an `ACK` message for the `200 OK` message. Having established the session, the two endpoints communicate directly, peer-to-peer, using a media protocol such as RTP [39]. However, this media session does not traverse the proxy, by design. When the conversation is finished, the UAC "hangs up" and generates a `BYE` message that the proxy forwards to the UAS. The UAS then responds with a `200 OK` which is forwarded back to the UAC.

SIP proxy authentication is an optional operation, typically done between a UA and its first hop SIP proxy server. While the example above shows a single SIP proxy along the path, in practice it is common to have multiple proxy servers in the signaling path. The message flow with multiple proxy servers is similar, except that the proxy authentication is usually only applicable to the first hop.

2.3 SIP Connection Management over TLS

SIP can operate over different transport protocols, both reliable and unreliable. Since TLS requires a reliable transport, all our evaluations for TLS use TCP transport. In general, a TCP connection is first established between endpoints, and then a TLS handshake occurs to negotiate the TLS session. Once the TLS session is established, the SIP signaling messages will be passed to the TLS layer and encrypted.

When a connection oriented transport such as TCP is used, the connection management policy needs to be defined. In a multi-hop SIP server network scenario, it is usually preferable to maintain a single long-lasting connection between two interconnected proxy servers. All SIP messages between the two proxy servers that go through the same existing connection can avoid the per-session connection handshake overhead. In contrast, if the proxy server is connected with a SIP UAC or UAS directly, the proxy typically has to establish separate connections with each of them since they are located on separate hosts.

3. TESTBED AND METHODOLOGY

3.1 OpenSIPS SIP Server

The SIP server we evaluated is Open SIP Server (OpenSIPS) version 1.4.2 [27], a freely-available, open source SIP proxy server. OpenSIPS is a fork of OpenSER, which in turn was a fork of SIP Express Router (SER) [20]. All these proxy servers are written in the C language, use standard process-based concurrency with shared memory segments for sharing state, and are considered to be highly efficient. These sets of server implementations represent the de facto open source version of SIP server, occupying a role similar to that of Apache for web server [4, 6, 13, 14, 16, 17, 24, 30, 42].

We made several modifications to OpenSIPS in order to support all of our test cases. In particular, we added a connection mode where OpenSIPS will establish a new connection to a UAS upon a new call, even if the UAS has the same IP address. This is needed to test the multiple connection mode between the proxy server and UAS using a limited number of UAS machines. We also added OpenSIPS options to to request TLS session reuse when it is acting as the TLS client, and OpenSIPS options to request for TLS mutual authentication when it is acting as the TLS server.

One unexpected parameter that initially prevented us from running high load tests with SIP proxy authentication is the "nonce index" value in OpenSIPS. It turns out that the default `MAX_NONCE_INDEX` value used to create nonce for proxy authentication is too small and could exhaust easily at high load. When the nonce could no longer be generated, authentication cannot proceed and the server will simply reject calls. We increased the default `MAX_NONCE_INDEX` value from $100,000$ to $10,000,000$. This change alone increased the throughput results dramatically, e.g., in the proxy chain mode the peak throughput with SIP proxy authentication is increased by close to an order of magnitude.

In configurations involving proxy authentication where a user database is required, we use MySQL-5.0.67 [26], which we populated with $10,000$ unique user names and passwords. The MySQL server runs on the same machine as the OpenSIPS server.

3.2 SIPp Client Load Generator

We use another freely available open-source tool, SIPp [19] to generate SIP traffic. SIPp allows a wide range of SIP scenarios to be tested, such as UAC, UAS and Third-Party Call Control (3PCC). We use the SIPp release from August 26th, 2008. We also added additional functionality to SIPp to accommodate all our test cases. Specifically, we added SIPp options to request TLS session reuse when it is acting as the TLS client and SIPp options to request TLS mutual authentication when it is acting as the TLS server. The TLS support library for SIPp is a statically-compiled version based on OpenSSL [28] release 0.9.8i (which is the latest release at the time of the compilation).

3.3 Hardware and Connectivity

The server hardware we use has 2 Intel Xeon 3.06 GHz processors with 4 GB RAM and 34 GB disk drives. However, for our experiments, we only use one processor because SIP performance under multiple processors or a multi-core processor is itself a topic that requires separate attention [42]. We use 10 client machines, six of which have 2 Intel Pentium 4 3.00 GHz processors with 1 GB RAM and 80 GB hard drives. The other four have 2 Intel Xeon 3.06 GHz processors with 4 GB RAM and 36 GB hard drives. The server and client machines communicate over copper Gigabit or 100Mbit Ethernet. The round trip time measured by the `ping` command from the client to the server is around 0.15 ms.

3.4 Software Platform

The server uses Ubuntu 8.04 with Linux kernel 2.6.24-19, OpenSSL 0.9.8.g, and oprofile 0.9.3. The clients use Ubuntu with either a 2.6.22 kernel or a 2.6.24 kernel. We encountered an SSL library failure at the SIPp load generator side when generating high loads. After examining the OpenSSL error queue in more detail, the `ERR_error_string` is found to be `error:1409F07F:SSL routines:SSL3_WRITE_PENDING: bad write retry`. A bug fix is found at [18]. This fix was submitted in 2003 but had not yet been incorporated into the OpenSSL release. We therefore recompile SIPp using OpenSSL version 0.9.8i source with this fix included. The OpenSIPS server machine uses the existing OpenSSL version 0.9.8g. The bug does not manifest itself there and keeping the original OpenSSL on the server makes profiling more convenient.

3.5 Workload and Performance Metrics

The workload is a standard SIP call flow provided by SIPp illustrated in Figure 2. There is no call hold time. Our main metrics are server throughput as reported by SIPp and server profile CPU events as reported by oprofile [29]. We also measure server CPU utilization. All our test runs last for 120 seconds after a 30-second warm-up time. All metrics are the average of three consecutive test runs.

3.6 Test Matrix and Evaluated Test Cases

We first group possible SIP server connection management configurations into four different deployment modes as shown in Figure 3.6.

1. Figure 3(a) shows the *proxy chain* mode, where the proxy server interconnects two other proxy servers in a chain fashion. This is intended to model, e.g., how two core SIP proxy servers of different service providers communicate. Only one connection is needed for each neighboring proxy server in this case.

2. Figure 3(b) shows the *outbound proxy* mode, where the proxy accepts multiple connections from UACs but only establishes a single outgoing connection with another proxy server. This configuration models how phones in an enterprise VoIP deployment would make calls external to the organization.

3. Figure 3(c) is the *inbound proxy* mode, where the proxy server under test accepts a single connection from an upstream proxy server and establishes multiple connections to individual UASes. This is the mirror of the outbound proxy configuration above, where incoming SIP traffic is routed to phones.

4. Figure 3(d), is the *local proxy* mode, where the proxy server under test connects UACs and UASes directly, and therefore accepts both incoming connections and creates outgoing connections simultaneously. This configuration is intended to model how phones in an enterprise deployment would communicate with each other.

SIP proxy servers usually support all these four modes of operation, thus this characterization is somewhat logical rather than physical. While in practice real proxy behavior will lie somewhere in the middle of these four extremes, the characterization lets us explore the design space fully.

For example, a SIP proxy operating in the proxy chain mode could well connect a number of different proxy pairs. It does not necessarily interconnect only a single pair of proxy servers. Similarly, an outbound proxy might connect to more than one upstream proxy. The four modes thus describe the full range of connection management behavior for SIP proxy servers, from completely persistent connections to a small set of nodes (the proxy chain mode) to non-persistent connections where each call requires a connection setup and teardown (the local proxy mode). In addition, the inbound and outbound cases distinguish where connections are passively accepted (the inbound proxy mode) vs. those that are created (the outbound proxy mode). To explore the applicable test matrix, we characterize five main configuration variables in our SIP-over-TLS tests: TLS connection management, TLS session reuse, TLS mutual authentication, TLS cipher suite and SIP proxy authentication. Note that the connection management configuration options also applies to TCP.

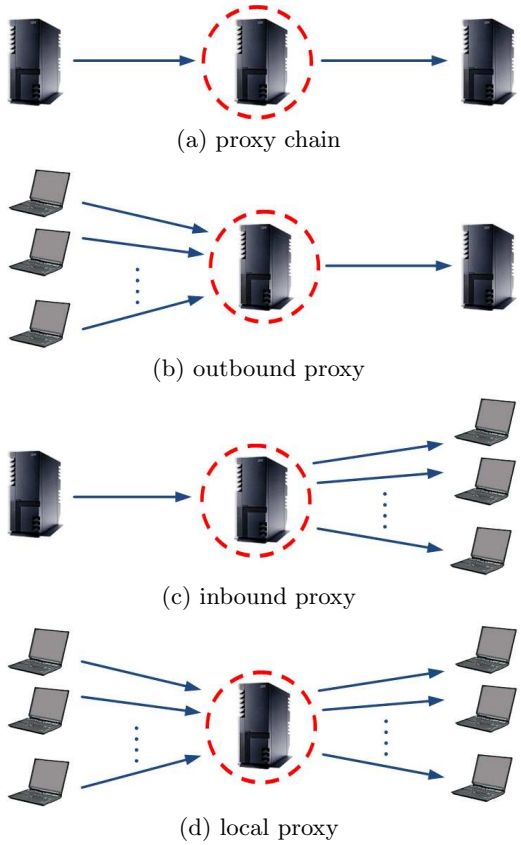


Figure 3: SIP Proxy Operation Modes

To relate connection management with other configuration parameters, we draw a unified logical component graph of the testbed as in Figure 4. The proxy server in the middle represents the server under test. The whole testbed is split into the left path and the right path, which consists of the left pair and the right pair of the logical UAC and UAS components, respectively. The applicable configuration options in each of the four connection management modes can then all be mapped into Table 1, where N/A indicates "Not

Configuration	TCP/TLS Multiple Connections		TLS Session Reuse		TLS Mutual Authentication		TLS Cipher Suite	SIP Proxy Auth.
	Left Path	Right Path	Left Path	Right Path	Left Path	Right Path		
Proxy Chain	N/A	N/A	N/A	N/A	N/A	N/A	any	Yes/No
Outbound Proxy	Yes	N/A	Yes/No	N/A	Yes/No	N/A	any	Yes/No
Inbound Proxy	N/A	Yes	N/A	Yes/No	N/A	Yes/No	any	Yes/No
Local Proxy	Yes	Yes	Yes/No	Yes/No	Yes/No	Yes/No	any	Yes/No

Table 1: Overall Test Matrix

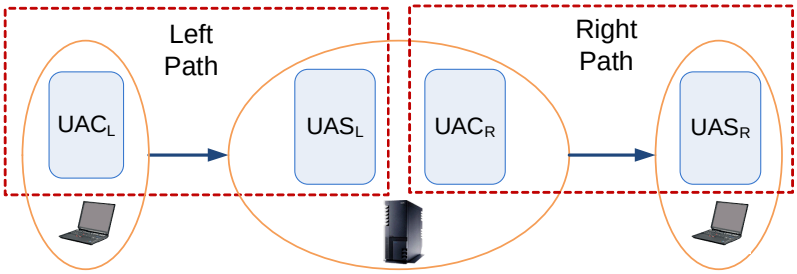


Figure 4: Logical Component Graph of SIP Testbed

Applicable".

Directly expanding the whole test space in Table 1 results in numerous configuration scenarios which are both intractable and unnecessary. We make the following decisions to narrow down the cases towards a workable test set. First, for TLS cipher suite, since the SIP standard [35] already specifies the mandatory `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite (abbreviated as TLS-AES) and a recommended `TLS_RSA_WITH_3DES_EDE_CBC_SHA` cipher suite (abbreviated as TLS-3DES), we focus on these two cipher suites only. In particular, since the impact differences between these two cipher suites are mainly on the bulk data encryption phase, we test both cipher suites only in the proxy chain mode which is specifically meant to examine the impact of TLS bulk data encryption. For all other three proxy modes, we test TLS-AES only. Second, we enable SIP proxy authentication only in the outbound proxy and local proxy modes, which is a common setting. Third, we test the TLS session reuse and TLS mutual authentication separately to understand each of their impacts. We configure appropriate certificates on both servers and clients in experiments which require them. Fourth, when both the left path and the right path can apply TLS session reuse or TLS mutual authentication, both paths have the same setting. These decisions reduce our test space for TCP and TLS to 16 configurations. Adding the two UDP Auth and UDP NoAuth settings, we have a total of 18 test configurations.

4. RESULTS AND ANALYSIS

Different proxy modes and configuration scenarios can incur significantly different overheads and result in very different limits on throughput. We start with the relatively simple proxy chain mode and then examine the more complex modes of outbound proxy, inbound proxy, and local proxy. For each of the 18 scenarios, we measure peak throughput and then use CPU profiles to understand and explain the performance costs.

4.1 Proxy Chain

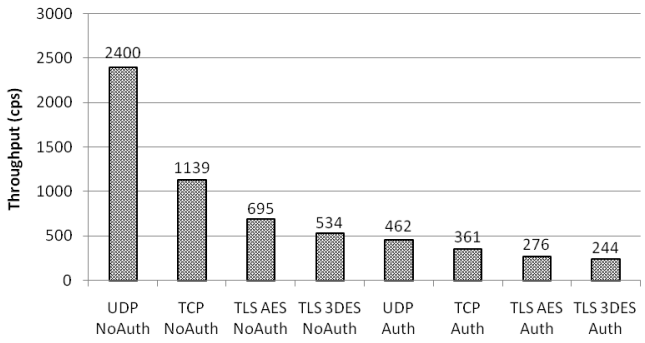


Figure 5: Peak Throughput: Proxy Chain

Figure 5 shows the peak throughput in calls per second (cps) for the proxy chain mode using several configurations. Each bar shows the performance for a different configuration. The first four bars have SIP proxy authentication disabled and the next four have SIP proxy authentication enabled. The tests include UDP, TCP only, TLS with the TLS-AES cipher suite, and TLS with the TLS-3DES cipher suite. Recall that in this mode, no connection setup overheads are incurred. The average CPU utilization ranges from 95% to 100% in all the peak test cases except for the UDP and TCP without authentication cases, which is about 70% and 85%, respectively. Note that not all the tests could reach full CPU utilization because there is not always quite enough client machines to fully load the testbed.

We see from Figure 5 that the peak throughput using TCP achieves about 47% of the throughput using UDP, when SIP proxy authentication is not used. When authentication is enabled, TCP provides 78% of the corresponding UDP performance. Adding TLS to the scenario results in even more substantial performance reductions. When SIP proxy au-

thentication is not enabled, TLS-AES achieves 60% of the corresponding TCP throughput, and TLS-3DES achieves 47% of the TCP throughput. When proxy authentication is enabled, TLS-AES achieves 76% of the corresponding TCP throughput and TLS-3DES achieves 68% of the TCP throughput.

While it would be convenient to simply attribute the extra overheads to the corresponding encryption algorithms, it turns out the reality is more complex. To better understand the overheads, we turn to the CPU profiles generated by oprofile. Our approach is to obtain a CPU profile of each configuration run at the same load level of 50 calls per second so that results across configurations can be compared meaningfully. As components are added (e.g., TLS vs. no TLS) or changed (AES vs. 3DES), the attendant CPU costs will manifest themselves in the profiles. This assumes costs scale relatively linearly with load and exhibit the same proportions at the peak as they do at 50 cps, which is not always the case. To test the accuracy of this assumption, we compare the observed peak throughputs with the ones extrapolated based on the CPU cycle costs observed. On average, the estimates match the observed peaks within 15 percent.

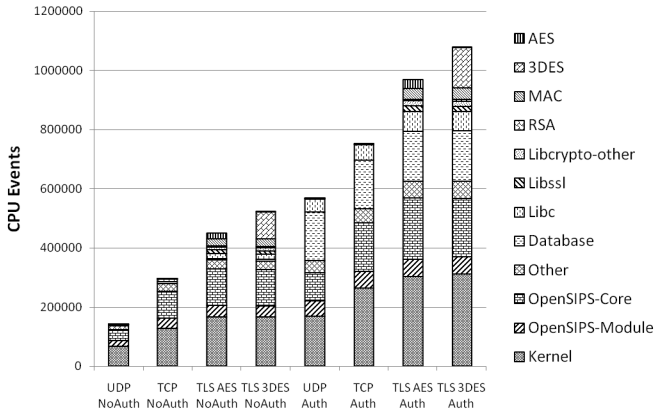


Figure 6: CPU Profile Cycle Costs: Proxy Chain (50 cps)

Figure 6 shows the number of non-idle CPU cycles consumed by the server in the proxy chain mode for each configuration during the test. We see that the total cost of the baseline UDP case without SIP authentication is about 144K CPU cycles. The most significant cost components are kernel (68K) which accounts for 47%, and the sum of OpenSIPS-Core and OpenSIPS-Model (54K), which contributes another 38% of the total cost. When TCP is used instead of UDP, the total costs increase 152K cycles or over 100%. Again most of the increase belongs to Kernel (60K) and the sum of OpenSIPS-Core and OpenSIPS-Module (71K).

We see that adding TLS-AES introduces another 50% of additional overhead, roughly 450K cycles vs. 300K cycles for the TCP case. TLS-3DES is similar, with roughly 525K cycles, and as would be expected, the differences in total cost between TLS-AES and TLS-3DES are almost solely contributed by the cost difference in cryptographic operations.

Half of the 150K increase from TCP to TLS-AES is directly contributed by TLS operations, and most of the remainder is relatively evenly shared by increases in Kernel and OpenSIPS-Core. Since 128 bits AES is less expensive than SHA-1, AES itself only adds about 19K cycles in cost; Message Authentication Code (MAC) overheads are higher at 25K cycles. MAC overheads are incurred by the bulk encryption algorithm, since each message is verified for authenticity using the MAC algorithms. MAC overheads are roughly equivalent regardless of the choice of AES or 3DES since we use SHA-1 in both cases. While 3DES is over 4X as expensive as AES (93K vs. 19K cycles), the relative difference between the two complete software stacks is only about 17% (525K vs. 450K). We expect AES to be faster since it is a more recent cipher than 3DES and was designed for performance. Other TLS overheads come from other components in the OpenSSL library. For example, in the TLS-AES case, there are other libcrypto functions (10K) and libssl (11K). Thus a non-trivial component of SSL overheads is from using the SSL mechanisms, such as allocating, freeing, maintaining, and looking up SSL session state.

Comparing the TCP case and the two TLS cases, the CPU profiles do not show the increases in kernel and OpenSIPS-Core centering on any specific functions. Between the two TLS cases themselves, the cost of Kernel and OpenSIPS-Core are quite similar.

The major difference when SIP proxy authentication is enabled is the additional database cost, which ranges from $16-29\%$ of the total cost in each case. When the database overhead is included, TCP will introduce 32% overhead over UDP. TLS-AES and TLS-3DES will incur an additional 30% and 44% over TCP, respectively. The rest of the cost contributions are similar to when SIP authentication is not enabled, because the authentication database functions are orthogonal to the TLS functions.

4.2 Outbound Proxy

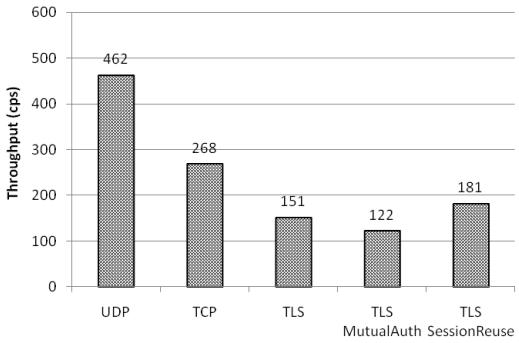


Figure 7: Peak Throughput: Outbound Proxy

Figure 7 shows the peak throughputs of the outbound proxy mode for several configurations. Recall that in the TCP or TLS cases of this mode, each call results in a new connection being established with the server, as opposed to the proxy chain mode above. Each bar again indicates a different configuration, namely UDP, TCP, TLS, TLS with mutual authentication, and TLS where session reuse occurs on each TLS connection. Each configuration has SIP authentication enabled. Since the choice of AES or 3DES only affects the bulk data encryption overheads, which we examined in Section 4.1, for simplicity we restrict our experiments

with TLS to use only AES for the remainder of this paper. The average CPU utilization in each case is around 90%. We see that the peak throughput in the TCP case is around 58% of the baseline UDP case. The TLS case is approximately 56% of the TCP case. Within the TLS cases, adding TLS mutual authentication reduces throughput about 20%, while enabling session reuse increases throughput about 20%.

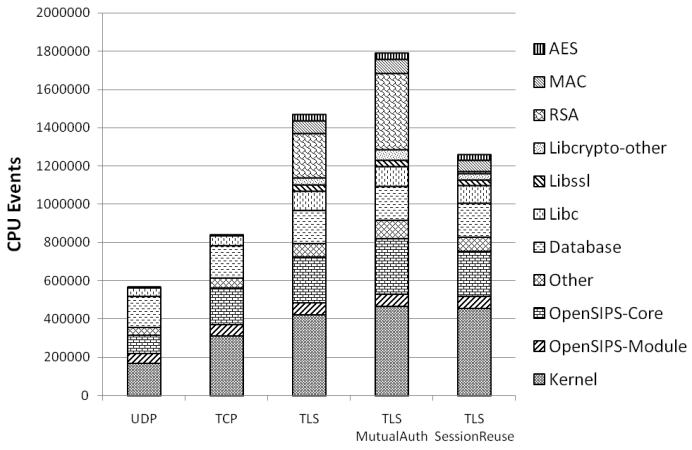


Figure 8: CPU Profile Cycle Costs: Outbound Proxy (50 cps)

Figure 8 shows the CPU profiles for the different outbound proxy configurations, again at the 50 calls per second load level. Using TCP introduces about 47% more or 271K of overheads compared to using UDP. Within this increase, Kernel costs contribute 144K, while OpenSIPS-Core and OpenSIP-Module contribute 102K. The remaining 25K is contributed by libc and other functions.

The use of TLS introduces 75% of additional overhead compared to the TCP case. TCP consumes about 840K cycles whereas TLS costs about 1,470K cycles. Much of this increase comes from RSA (233K cycles) because in this configuration the proxy needs to perform one of the most costly operations in the TLS handshake: RSA decryption of the `pre_master_secret` using its private key. Another major component of the increase is from MAC processing (65K cycles), which is not only used to verify the encrypted messages but also to verify the server certificate and construct the `master_secret`. Other OpenSSL overheads such as libssl (34K) and other libcrypto functions (36K) also contribute.

Enabling TLS mutual authentication incurs about 1,790K cycles or an additional 330K over the baseline TLS, most of which comes from increased RSA costs (160K). Recall in this case the server requests the client's certificate which the server verifies using RSA public key decryption. In addition, the server performs another RSA public key decryption for the client's certificate verification message and also verifies the certificate using the MAC algorithm. Indeed, we see MAC costs increase by 10K cycles when mutual authentication is used. Kernel costs also increase by 45K cycles, presumably due to additional network packets transmitted and context switches between user and kernel space.

However, enabling TLS session reuse reduces the overhead by 15% compared to the baseline TLS case, or by about 200K cycles. Most of this overhead is explained by the reduction in RSA costs, which shrink from 233K cycles to only 10K cycles. This is because in the session reuse case, no key exchange and certificate verification is required. MAC costs remain, however, since new cryptographic keys are still computed for data encryption.

It is worth noting that the TLS mutual authentication test above also includes SIP proxy authentication. While TLS mutual authentication is used to authenticate and authorize "client systems", SIP proxy authentication is used to authenticate and authorize "users".

4.3 Inbound Proxy

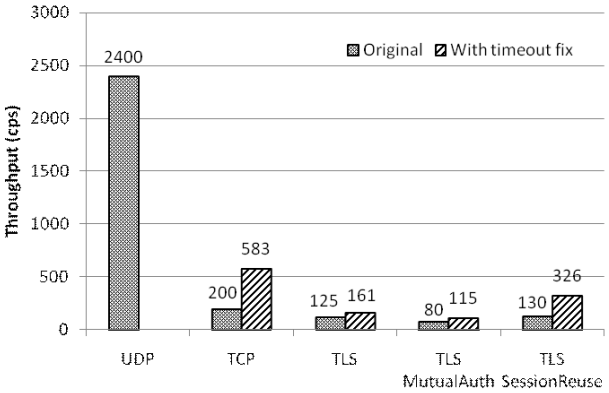


Figure 9: Peak Throughput: Inbound Proxy

Figure 9 shows the peak throughput of the inbound proxy mode, where SIP proxy authentication is not enabled. The figure shows two versions of OpenSIPS: the original version and one with a modification we developed, denoted "with timeout fix" in the graph. We start by explaining the performance problem we discovered and how we solved it.

We examined the original OpenSIPS CPU profile under the peak throughput for TCP and TLS. Surprisingly, we found that 50% of the CPU cycles in the TCP case and 20% percent of the CPU cycles in the TLS case are spent in two functions, `tcp_main_loop` and `tcp_receive_loop`. More detailed profiling reveals that the overhead in the two functions are primarily the cost of two timeout mechanisms used to close the TCP connections which are no longer in use. In the inbound proxy case, the timeout mechanism becomes prominent because the UAS in our tests does not close the TCP/TLS connection when the call is over. There can be thousands of simultaneous TCP connections existing in the TCP connection table. The current server code calls a `timeout` function every time the `epoll` mechanism returns when events are detected. Since the connection expiration time is not linked to the corresponding hash key, during each call to the `timeout` function, the entire TCP connection hash table is traversed . Therefore, at high loads when the hash table has thousands of entries, the time spent in the timeout function becomes much larger than that of the case under lower load.

We applied a fix to the existing OpenSIPS TCP connection timeout mechanism. Observing that the timeout is based on a time tick with one second resolution, it makes no sense to enter the timeout function more than once per second. We therefore added a time tick check before calling

the timeout function. If the program has called the timeout function during the current time tick value already, then it will not enter the timeout function until the time tick value is advanced. This simple fix turned out to have a drastic effect on performance involving TCP and TLS, as shown in Figure 9.

As can be seen, the TCP case and the TLS with session reuse scenario enjoy the most obvious boosts in throughput, by about 200% and 150% respectively. For example, in the TCP inbound proxy test case, the contribution of the two timeout functions to the total overhead reduces from 50% to a negligible 0.6%, and the total cost drops by 73%. In addition, kernel costs shrink by 43%. CPU utilization at the 200 calls per second load level reduces from 95% to 20%. The CPU utilizations at the peak throughput values with the timer fix are in the range of 80% to 90%.

The other two scenarios, TLS and TLS with mutual authentication, also see performance increases but the differences are less dramatic. The reason is that in the latter two scenarios, the proportion of cryptographic overheads take a greater part of the total cost, so reducing the OpenSIPS and kernel overheads has a smaller impact.

From Figure 9, we see that the peak throughput with TCP is about 24% of the UDP case. The peak throughput of TLS is approximately 28% of the TCP case. Within the TLS cases, adding TLS mutual authentication reduces throughput by 29%, while enabling session reuse increases throughput by 100%.

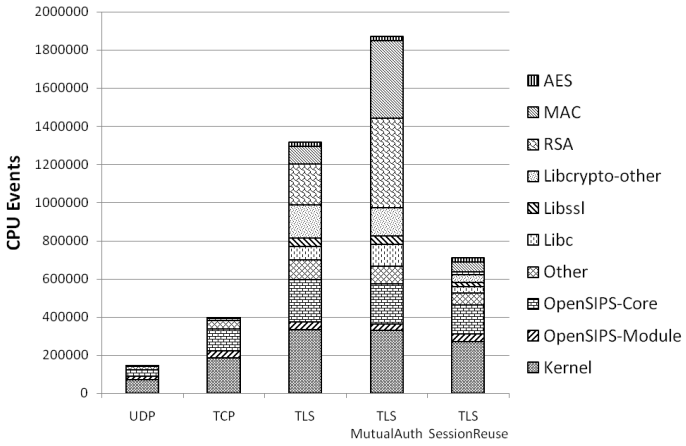


Figure 10: CPU Profile Cycle Costs: Inbound Proxy (with Timeout Fix)

Figure 10 shows the CPU profiles for the several inbound proxy configurations where the timeout fix has been applied. In general, using TCP incurs 174% (250K) of additional overhead compared to using UDP, 118K of which comes from increase in Kernel and 94K from increases in OpenSIPS-Model and OpenSIPS-Core. The remainder comes from libc (8K) and other functions (30K). The use of TLS introduces over 233% of additional overhead compared to the TCP case (1,315K cycles vs. 394K). 212K cycles are contributed by RSA, 173K by other libcrypto processing, 93K by MAC processing, 44K by libssl, and 23K by AES. Kernel overheads increase by 150K and OpenSIPS-Core by 110K.

Enabling mutual authentication incurs an additional 42% overhead (550K cycles) over the baseline TLS. The majority of that increase comes from RSA (260K). MAC processing is also increased by 310K.

Enabling TLS session reuse reduces costs by 46% compared to the base TLS case, with total costs falling from 1,315K to 710K or about 600K cycles. Reduced RSA processing contributes 200K of those reductions; other libcrypto costs drop by 135K; MAC overheads are reduced by 40K; libssl costs shrink by 20K.

In this configuration, the main RSA costs in the normal TLS case come from the proxy verifying the UAS's certificate and the proxy encrypting the `pre_master_secret` to be sent to the UAS. The additional increase in RSA overheads in the mutual TLS configuration is mainly because the proxy needs to sign the client authentication message using its private key.

An interesting observation from this figure is the cost of MAC functions, which are substantially higher than in the previous proxy scenarios. This is because the proxy in the inbound mode acts as TLS client and needs to verify the certificates presented by the UAS, which was not present in the outbound mode. In addition, in the mutual TLS case, the inbound proxy needs to perform RSA encryption using its own private key and to sign the certificates using the MAC algorithm. These overheads are exhibited in the profiles. Furthermore, in the TLS with session reuse case, the MAC costs are significantly reduced, indicating that a large amount of the MAC cost is associated with the RSA key exchange phase, rather than during the bulk data encryption.

4.4 Local Proxy

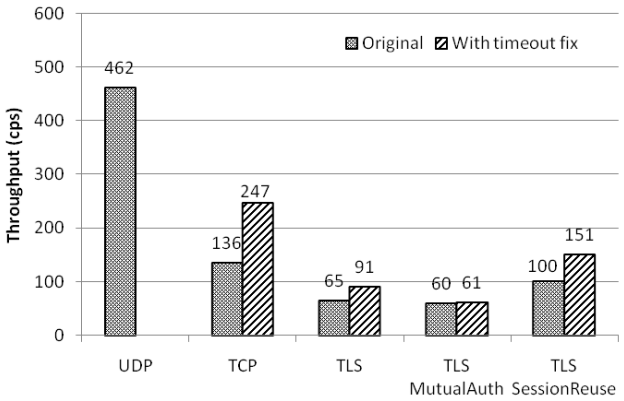


Figure 11: Peak Throughput: Local Proxy

Figure 11 shows the peak throughputs of various configurations in the local proxy mode, both with and without the timeout fix described in Section 4.3, and with SIP authentication enabled. We see the timeout fix has a substantial impact on performance for both the baseline TCP case and for TLS when session reuse is enabled, where TCP overheads are significant. The timeout fix makes less of an impact on the other TLS cases because in those cases the TLS overheads account for a larger proportion of the total cost. For the remainder of this Section, we focus our analysis on the configurations where the timeout fix is applied.

The average CPU utilizations in the four configurations

with the timeout fix are between 80% to 90%. We see that the peak throughput with TCP is around 53% of the UDP case, while the peak throughput with TLS is approximately 37% of the TCP case. Within the TLS cases, adding TLS mutual authentication reduces throughput by 33%, while enabling session reuse increases throughput by 66%.

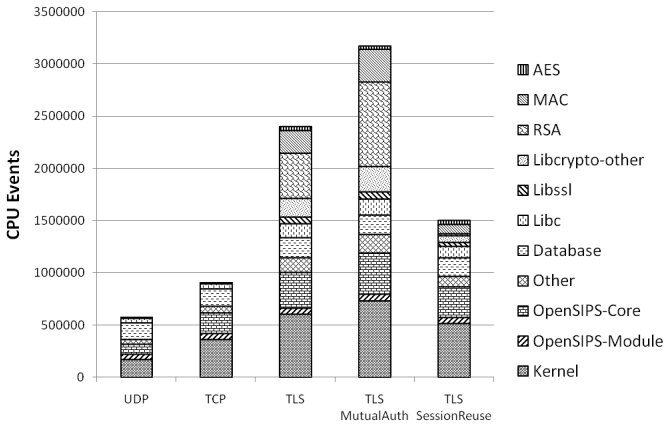


Figure 12: CPU Profile Cycle Costs: Local Proxy (with Timeout Fix)

Figure 12 shows the CPU profile results for the local proxy mode with the timeout fix. In general, the use of TCP incurs 58% of additional overhead compared to the baseline UDP case. 186K of this is contributed by Kernel, 108K by OpenSIPS-Core and OpenSIPS-Module, 10K by libc and 30K by other functions. Using TLS introduces over 166% of additional overhead compared to the TCP case. Total cycles increase by 1,500K from 900K to 2,400K. RSA contributes 434K to that increase, followed by kernel overheads 240K, MAC processing 219K, other libcrypto functions 174K, OpenSIPS-Core 140K, libssl 67K, and AES 36K.

Enabling TLS mutual authentication incurs an additional 32% overhead over the baseline TLS, increasing total costs about 800K from 2,400K to 3,170K. Additional RSA overheads contribute 375K of the increase, 125K from kernel, 100K from MAC, 70K from libcrypto, 45K from OpenSIPS-Core, and 5K from libssl.

Enabling TLS session reuse reduces the cost relative to the baseline TLS case by 38%. Cycles shrink by 900K from 2,400K to 1,500K. RSA savings contribute 415K to the difference, followed by MAC 130K, other libcrypto functions 110K, kernel 80K, OpenSIPS 50k, libssl 25k.

The MAC cost is significantly reduced in the TLS with session reuse case, indicating that a large amount of the MAC cost is associated with the RSA public key exchange phase, as discussed in the inbound proxy case in Section 4.3.

5. RELATED WORK

SSL/TLS performance has been studied by a number of researchers. However, almost all these studies are based on SSL/TLS Web servers. Apostolopoulos et al. [5] found that the overhead due to TLS can reduce the number of HTTP transactions handled by up to two orders of magnitude. Kant et al. [21] investigated the architectural impact of SSL, and concluded that the use of SSL increases the computational cost of transactions by a factor of $5-7$. Zhao et al. [44] provided an oprofile-based anatomy of SSL processing for an SSL Web server. They found that about 70% of the total processing time of an HTTP over SSL transaction is spent in SSL processing. Coarfa et al. [10] measured the difference of TLS server throughput by selectively replacing TLS operations with no-ops, instead of using a CPU profiler. Their results show that RSA computations are the single most expensive operation in TLS, which accounts for $13-58\%$ of the total time spent under different available server CPU cycles and workload conditions.

Zeng and Cherkaoui [43] studied the performance of TLS on a Common Open Policy Service (COPS) over TLS environment. The results of their study showed that establishing a COPS over TLS session took about a thousand times as much as needed for a pure COPS connection without TLS.

Many researchers have studied SIP server performance, albeit without TLS. Schulzrinne et al. presented SIPstone [40], a suite of SIP benchmarks for measuring SIP server performance on common tasks. Cortes [12] measured the performance of four different stateful SIP proxy server implementations over UDP and reported throughput results from $90-700$ cps. Nahum et al. [16, 24] showed experimental performance results of the OpenSER SIP server under different scenarios including stateful and stateless proxying, TCP and UDP transport, with and without SIP proxy authentication. Their results indicate that any of these configurations can affect performance by a factor of $2-4$. Their evaluated SIP-over-TCP scenario corresponds to the TCP single connection or the proxy chain mode in this paper. Oho and Schulzrinne [25] studied the performance of the SIPd [38] SIP server over the UDP and TCP transports. Their TCP tests include the multiple connection mode between the SIP proxy and the UA similar to the local proxy mode of this paper. Ram et al. [30] provided more understanding of the impact of TCP on SIP server performance using OpenSER. They show that a substantial component of the performance loss from using TCP is due to the process architecture in OpenSER and provide improvements. Wright et al. [42] studied the performance of SIP servers on multi-core systems. They proposed and evaluated several optimizations to improve scalability up to eight cores.

Cha et al. [9] described a study of SIP with TLS, DTLS and IPSec over TCP, UDP and SCTP. However, the work is based on ns-2 [1] simulation and the scope of the evaluation is on call setup delay in a two-hop SIP proxy scenario with background traffic. Thus the focus is on delay as a function of packet exchanges rather than server CPU overheads. Kim et al. [23] also measured the call setup delay (along with voice quality metrics such as mean option score) of a SIP-based VoIP system implementation which contains both TLS and S-MIME. But it is not clear what the software and hardware used are, or what the call request rate during the measurement is.

The most relevant work we found is from Salsano et al. [36] who measured the throughput performance and processing cost of SIP proxy server over UDP, TCP and also TLS. Their test cases for stateful SIP proxy servers represent four of the 18 scenarios that we look at, essentially the UDP NoAuth, UDP Auth, TCP Auth, and TLS Auth configurations, all in the proxy chain mode. The total cost ratios of these four scenarios in their work are 1:1.44:1.52:1.54, while the corresponding ratios from our results are 1:4:5.2:6.7. These

numbers are not directly comparable because of the different software and hardware platforms used in the two sets of experiments. Salsano et al. used their own open source SIP server implemented in Java using a 300 MHz Pentium machines running either Linux or Windows 98/2000. We use contemporary hardware and standard open-source software implemented in C. As a result, the peak performance of the two testbeds are also dramatically different. For example, in the basic UDP NoAuth scenario, the peak throughput on their testbed is 21 cps, compared to 2,400 cps on ours, a factor of 100 difference in performance.

One approach to reducing security overheads is to use a hardware crypto accelerator, e.g., Sun's Crypto 6000 card [11]. While this can improve performance (e.g., the card claims 13,000 1024-bit RSA operations per second), the cards tend to be expensive (e.g., the list price for the board was $9,950 at the time of this writing). More importantly, in many cases, much of the overhead we observed was in the OpenSSL software libaries themselves (e.g., libssl, libssl-other), rather than the crypto algorithms (libcrypto). Crypto acceleration hardware will not help with these overheads.

6. CONCLUSIONS

Insecure UDP-based signaling is one major reason that exposes SIP-based services to many common security threats. We have evaluated and analyzed the impact of using TLS as a transport on SIP server performance versus the standard approach of SIP-over-UDP. Using an experimental testbed with the OpenSIPS server, OpenSSL, Linux, and an Intel-based server, we show that the performance with TLS can be reduced significantly. We use application, library, and kernel profiling to illustrate where different costs are incurred (e.g., extra RSA overheads when mutual authentication is used) and how they can be avoided (i.e., RSA costs are nearly eliminated when session reuse is effective).

In the best case, the baseline UDP performance is about three times that with TLS (the proxy chain mode); in the worst case, UDP is 17 times the performance than with TLS (the local proxy with TLS and mutual authentication). The performance results depend primarily on whether and how frequent TLS connection establishment is performed, since TLS session negotiation incurs expensive RSA public key operations. In turn, session negotiation depends on how the SIP proxy is deployed (as an inbound, outbound, or local proxy) and how TLS is configured (with mutual authentication or session reuse). Bulk encryption costs such as 3DES or AES, in contrast, are minimal, typically no more than seven percent.

Implementation plays a role as well. We found several performance bugs in OpenSIPS and OpenSSL, despite the fact that they have mature code bases and large numbers of users. When fixed, performance improved in some cases from a few times up to an order of magnitude.

Network operators considering deploying SIP over TLS will need to consider the extra resources required to provide the same service quality as would be the case with UDP. Costs can be reduced by maximizing the potential for persistent TLS sessions, which avoid heavy connection setup costs. These lessons may be appropriate for other protocols that use TLS, especially if they tend to have short messages.

Acknowledgments

The authors would like to thank the anonymous reviewers for insightful and detailed comments which helped improve this paper. Charles Shen would like to acknowledge Dr. Arata Koike of NTT for useful discussions.

7. REFERENCES

[1] ns-2 simulator. `http://www.isi.edu/nsnam/ns/`.

[2] SIP forum. `http://www.sipforum.org`.

[3] VoIP security alliance. `http://www.voipsa.org`.

[4] A. Acharya, X. Wang, and C. Wright. A programmable message classification engine for Session Initiation Protocol (SIP). In *Proc. of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 185–194, Orlando, FL, December 2007.

[5] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Proc. of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, NY, March 1999.

[6] V. Balasubramaniyan, A. Acharya, M. Ahamad, M. Srivatsa, I. Dacosta, and C. Wright. Servartuka: Dynamic distribution of state to improve SIP server scalability. In *Proc. of the 28th International Conference on Distributed Computing Systems*, pages 562–572, Beijing, China, June 2008.

[7] D. Butcher, X. Li, and J. Guo. Security challenge and defense in VoIP infrastructures. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(6):1152–1162, November 2007.

[8] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitima, and D. Gurle. Session Initiation Protocol (SIP) extension for Instant Messaging. RFC 3428, December 2002.

[9] E. Cha, H. Choi, and S. Cho. Evaluation of security protocols for the Session Initiation Protocol. In *Proc. of the 16th International Conference on Computer Communications and Networks (ICCCN)*, Honolulu, HI, August 2007.

[10] C. Coarfa, P. Druschel, and D. Wallach. Performance analysis of TLS Web servers. In *Proc. of the Internet Society Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2002.

[11] Oracle Corporation. Sun crypto accelerator 6000 PCIe card. `http://www.oracle.com/us/products/servers-storage/networking/031146.htm`.

[12] M. Cortes, J. Ensor, and J. Esteban. On SIP performance. *IEEE Network*, 9(3):155–172, Nov 2004.

[13] I. Dacosta, V. Balasubramaniyan, M. Ahamad, and P. Traynor. Improving authentication performance of distributed SIP proxies. In *Proc. of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–11, Atlanta, GA, July 2009.

[14] I. Dacosta and P. Traynor. Proxychain: Developing a robust and efficient authentication infrastructure for carrier-scale VoIP networks. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, June 2010.

[15] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.

[16] E. Nahum and J. Tracey and C. Wright. Evaluating SIP server performance. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):349–350, June 2007.

[17] J. Fabini, N. Jordan, P. Reichl, A. Poropatich, and R. Huber. "IMS in a bottle": Initial experiences from an OpenSER-based prototype implementation of the 3GPP IP multimedia subsystem. In *Proc. of the International Conference on Mobile Business*, page 13, Copenhagen, Denmark, June 2006.

[18] RT for openssl.org. Ticket no. 598. `http://rt.openssl.org/Ticket/Display.html?id=598\&user=guest\&pass=guest`.

[19] R. Gayraud and O. Jacques. SIPp. `http://sipp.sourceforge.net`.

[20] IPTel.org. SIP express router (SER). `http://www.iptel.org/ser`.

[21] K. Kent, R. Iyer, and P. Mohapatra. Architectural impact of secure socket layer on Internet servers. In *Proc. of International Conference on Computer Design (ICCD)*, pages 7–14, Austin, TX, October 2000.

[22] A. Keromytis. Voice over IP: Risks, threats and vulnerabilities. In *Proc. of the Cyber Infrastructure Protection (CIP) Conference*, New York, NY, June 2009.

[23] J. Kim, S. Yoon, H. Jeong, and Y. Won. Implementation and evaluation of SIP-based secure VoIP communication system. In *Proc. of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Shanghai, China, December 2008.

[24] E. Nahum, J. Tracey, and C. Wright. Evaluating SIP proxy server performance. In *Proc. of 17th International Workshop on Networking and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Urbana-Champaign, IL, June 2007.

[25] K. Ono and H. Schulzrinne. One server per city: Using TCP for very large SIP servers. In *Proc. of the 2nd International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 133–148, Heidelberg, Germany, October 2008.

[26] The MySQL Project. MySQL database server. `http://www.mysql.org`.

[27] The OpenSIPS Project. The open SIP server. `http://www.opensips.org`.

[28] The OpenSSL Project. The OpenSSL library. `http://www.openssl.org`.

[29] The OProfile Project. OProfile. `http://oprofile.sourceforge.net`.

[30] K. Kumar Ram, I. Fedeli, A. Cox, and S. Rixner. Explaining the impact of network transport protocols on SIP proxy performance. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 75–84, Austin, TX, April 2008.

[31] Light Reading. VoIP security: Vendors prepare for the inevitable. *VoIP Services Insider*, 5(1), January 2009.

[32] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison Wesley, 2000.

[33] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, April 2006.

[34] R. Rivest. The MD5 Message-Digest Algorithm . RFC 1321, April 1992.

[35] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[36] S. Salsano, L. Veltri, and D. Papalilo. SIP security issues: the SIP authentication procedure and its processing load. *IEEE Network*, 16(6):38–44, Nov/Dec 2002.

[37] B. Schneier. *Applied Cryptography (2nd Edition)*. John Wiley and Sons, Inc., New York, NY, 1996.

[38] H. Schulzrinne. SIPd. `http://www.cs.columbia.edu/IRT/cinema`.

[39] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[40] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone-benchmarking SIP server performance. `http://www.sipstone.com`.

[41] X. Wang, R. Zhang, X. Yang, X. Jiang, and D. Wijesekera. Voice pharming attack and the trust of VoIP. In *Proc. of the 4th international conference on Security and privacy in communication netowrks*, pages 1–11, Istanbul, Turkey, September 2008.

[42] C. Wright, E. Nahum, D. Wood, J. Tracey, and E. Hu. SIP server performance on multicore systems. *IBM Journal of Research and Developement*, 54(1), February 2010.

[43] Y. Zeng and O. Cherkaoui. Performance study of COPS over TLS and IPsec secure session. In *Proc. of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 133–144, Montreal, Canada, October 2002.

[44] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of SSL processing. In *Proc. of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 197–206, Austin, TX, March 2005.