The Impact of TLS on SIP Server Performance

Charles Shen*, Erich Nahum†, Henning Schulzrinne*, Charles P. Wright†

*Columbia University Computer Science Department
†IBM T.J. Watson Research Center

Columbia University Department of Computer Science Technical Report CUCS-022-09

Abstract

This report 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 nearly 20 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 or 3DES, in contrast, tends to be small.

Network operators deploying SIP over TLS should attempt to maximize the persistence of secure connections, and will need to assess the server resources required. To aid them, we provide a measurement-driven cost model for use in provisioning SIP servers using TLS. Our cost model predicts performance within 15 percent on average.

I. Introduction

Session Initiation Protocol (SIP) [34] 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, service abuse [2], [17], [7]. One of the main reasons that permit for 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 [4]). It has also been shown that even commercial VoIP services may be prone to large-scale voice pharming [40], where victims are directed to fake interactive voice response systems or human representatives for sensitive information.

Transport Layer Security (TLS) [8], [9] is a widely used Internet security protocol occupying a layer between the application and the reliable TCP transport. There is also a Datagram TLS (DTLS) [31] protocol that provides similar security functionalities but runs over the unreliable UDP transport. The current SIP specification [34] lists TLS as a standard method to secure SIP signaling. Various other organizations and industrial consortiums have also suggested or mandated the use of TLS for SIP signaling. For example, the SIP Forum [1] mandated TLS for interconnecting enterprise and service provider SIP networks in its specification document.

However, while interest in securing SIP is growing [28], 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 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 report makes the following contributions:

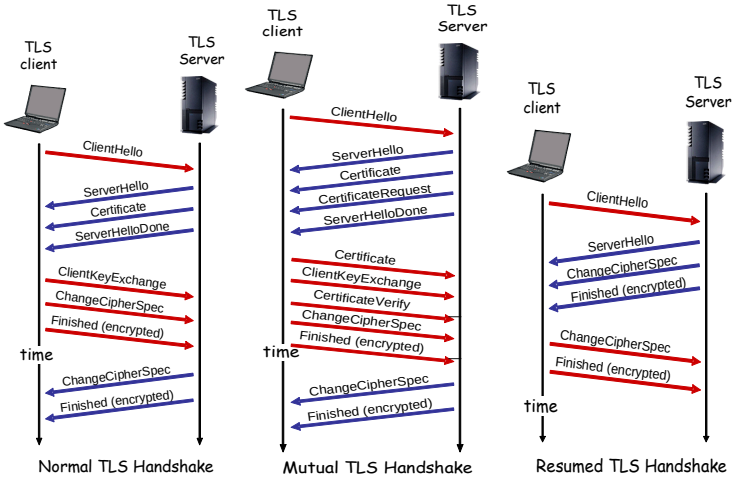
- We present an experimental performance study of the impact of using TLS on SIP servers. Our study is conducted using OpenSIPS with OpenSSL on Linux on an Intel-based server. We evaluate the cost of TLS under four SIP proxy usage scenarios: proxy chain, outbound proxy, inbound proxy, and local proxy. We show that using TLS can reduce performance by up to a factor of nearly 20 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 RSA [33] overheads when mutual authentication is used) and how they can be reduced (e.g., RSA costs disappear when session reuse is performed). They also show some results unique to SIP (e.g., bulk crypto costs of AES [22] or 3DES [19] are small), and how some overheads are due to mechanisms (e.g., kernel overhead, SSL state management) rather than simply crypto algorithms such as RSA or AES.

- We provide a cost model to aid network administrators that are considering transitioning to SIP over TLS. The cost model estimates server resource costs of TLS to help provisioning and dimensioning of servers. Our cost model accurately predicts performance within 15 percent on average.

Previous studies on TLS performance have either focused on TLS for Web servers [3], [5], [15], [42] or policy-based network management (COPS) [41]. 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 proxies can incur client-side TLS costs since they can act as clients to other servers. 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. Implementation issues can also be significant; we found several performance problems in OpenSIPS and OpenSSL, despite the fact that they are widely used and relatively mature.

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 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 II provides some background on TLS and SIP. Section III describes the experimental testbed used for our experiments. Section IV presents our results in detail. Section V develops our cost model. Section VI describes related work and we conclude in Section VII. Appendix A provides additional background on security and cryptography in TLS. Appendix B provides a few more operating system configuration modifications. Appendix C presents the mapping tables between function names and oprofile results used in our profiling analysis. Appendix D discusses a performance fix for the proxy software in establishing TLS connections.



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

Fig. 1. TLS Handshake Message Flows

II. Background

For space reasons, we assume the reader is familiar with basic security concepts of confidentiality, integrity, and authenticity, and how they are provided using public key and symmetric key cryptography. We assume the reader is generally aware of public key algorithms such as RSA; symmetric key ciphers such as AES, and 3DES; and signed message authentication codes such as MD5 and SHA-1. More detail on cryptography can be found in appendix A and [36].

A. TLS Operation Overview

For space reasons, we provide a very brief description of the TLS protocol. For more details, please see [9], [30]. We focus on the aspects relevant to our study, namely session establishment and its attendant 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.

All the algorithms used in a TLS session, including those for key exchange, bulk data encryption and message digest, are specified by a cipher suite. As an example, TLS_RSA_WITH_AES_128_CBC_SHA is one cipher suite indicating that RSA

public key algorithm is used for shared secret key exchange and authentication; 128-bit AES in CBC mode is used for bulk data encryption; and SHA-1 is used as the message digest algorithm to compute the message authentication code (MAC).

The normal message flow in the TLS handshake phase is illustrated in Figure 1(a). First the client initiates the handshake with a `ClientHello` message. This message contains the protocol version, the cipher suite and compression methods that the client supports and a random number and timestamp to prevent replay attacks. The server responds with a `ServerHello` message, which specifies the protocol version and the cipher suite and compression methods that the server chooses to use among those proposed by the client. The `ServerHello` message also contains a timestamp and random number as part of the keying material, and optionally a `session_id` which the client can later use to resume the session. The server then sends the `Certificate` message which is the server's X.509 certificate containing its public key and optionally a chain of certificates belonging to the authorities in the certificate hierarchy. The following `ServerHelloDone` message indicates the server has sent all message in this stage. Upon receiving the server's certificate, the client authenticates the server by verifying its certificate using the CA's public key. The client then generates a `pre_master_secret`, and encrypts it using the server's public key obtained from the server's certificate. This encrypted `pre_master_secret` is sent in the `ClientKeyExchange` message to the server. 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 addition, the client and server also exchange the `ChangeCipherSpec` message, which indicates that the sender has switched to the newly negotiated algorithms. Finally, the `Finished` message contains a MAC digest of the negotiated `master_secret` and the concatenated handshake message that have been sent to the other party. The `Finished` message is used to ensure the integrity of the handshake.

In the 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, shown in Figure 1(b), which allows what is called mutual authentication. 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.

Cryptographic operations can be costly. 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 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 share the same set of algorithms and the `master_secret`, but each with a different symmetric key 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 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.

B. 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 all 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

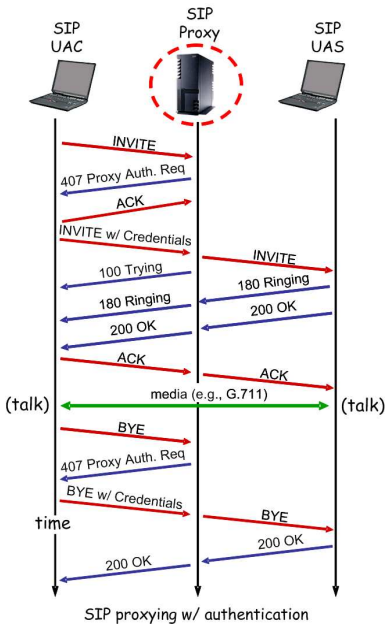


Fig. 2. SIP Stateful Proxying with Authentication

different levels of session state information. Therefore, we focus on stateful SIP proxying in this report. Figure 2 shows a typical message flow of stateful SIP proxying with authentication enabled.

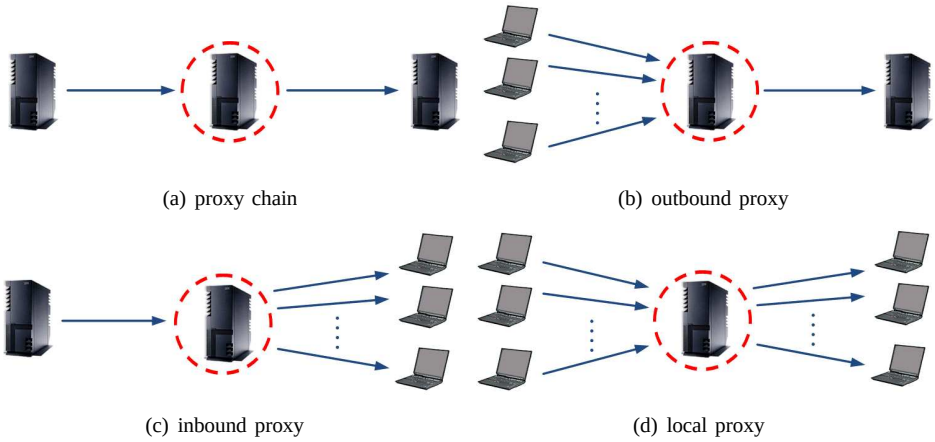
Two SIP UAs, designated by User Agent Client (UAC) and User Agent Server (UAS) represent the caller and callee of a multimedia session. The hashed circle around the proxy indicates that this is the server that we are measuring ("system under test"). In this example, the UAC wishes to establish a session with the UAS and sends an `INVITE` message to the proxy. The proxy server enforces an optional proxy authentication feature 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 digest algorithm). The UAC then retransmits the `INVITE` message with the generated credentials in the `Authorization` header. After receiving and verify 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 ring 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 [38]. 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 proxies will be similar, except that the proxy authentication is usually only applicable to the first hop.

C. Connection Management with SIP/TLS

SIP can operate over different transport protocols, both reliable and unreliable. Since TLS requires a reliable transport and TCP is the dominant reliable transport protocol in the Internet, all our evaluations use TCP. 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 generally 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. Given these observations, we group the possible SIP server connection management configurations into four different



(a) proxy chain (b) outbound proxy

(c) inbound proxy (d) local proxy

Fig. 3. SIP Proxy Operation Modes

modes as shown in Figure II-C. Figure 3(a) shows the *proxy chain* mode, where the proxy server interconnects two other proxy servers in a chain fashion. Only one connection is needed for each neighboring proxy server in this case. Figure 3(b) shows the *outbound proxy* scenario, where the proxy accepts multiple connections from UACs but only establishes a single outgoing connection with another proxy server. Figure 3(c) is the *inbound proxy* scenario, where the proxy server under test accepts a single connection from an upstream proxy server and establishes multiple connections to individual UASes. In Figure 3(d), the proxy server under test connects UACs and UASes directly, and therefore accepts both incoming connections and creates outgoing connections simultaneously.

SIP proxies usually support all these four modes of operation, thus this characterization is somewhat logical rather than physical. For example, a SIP proxy operating in the middle of a proxy chain does not necessarily interconnect only a single pair of proxy servers; it could well connect a number of different proxy pairs. 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 proxies, 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 case) vs. those that are created (the outbound case). While in practice real proxy behavior will lie somewhere in the middle of these extremes, the characterization lets us explore the design space fully.

Since we run TLS over TCP, the connection management scenarios described for TCP is equally applicable to the TLS case, but with one addition: the session reuse case. Thus there are three possible options for TLS. First is creating a new session from scratch, requiring both a new TCP connection and a new TLS session. Second is using a persistent existing session, with an established TLS session and TCP connection. Third is re-using an earlier TLS session, requiring a new TCP connection but performing TLS session reuse rather than a full new TLS session.

III. Experimental Methodology

Here we discuss the software and hardware utilized in our experiments. We also present any software tuning performed. Several subtle software bugs were exposed by driving the system with high loads that had a large number of TCP or TLS connections. We also describe the necessary changes and fixes we made where appropriate. Additional information about operating system parameter configuration is provided in Appendix B.

A. Test Matrix and Evaluated Test Cases

The biggest advantage of differentiating the four SIP server connection modes (chain, outbound, inbound, local proxy) as in Section II-C is to reduce the complexity and reveal actual contributing components of the system in different scenarios. Indeed, each of the first three connection modes allows us to examine a different aspect of the system in terms of TLS cost evaluation. For the proxy chain scenario, since there is no additional connection establishment cost once the signaling has started, it allows us to solely evaluate the cost impact incurred in TLS bulk data encryption. The outbound and inbound proxy scenarios include per-session connection management, therefore allowing us to assess the additional cost impact associated with the TLS handshake phase, where the proxy server acts as the TLS client side and the TLS server side, respectively. Finally, the forth connection mode, local proxy mode, gives us an overall view combining all the aspects involved in the first three scenarios.

TABLE I
OVERALL TEST MATRIX

Configuration	TCP/TLS Multiple Connections		TLS Session Reuse		TLS Mutual Authentication		TLS Cipher Suite	SIP UAC Auth.
	Left	Right	Left	Right	Left	Right		
Proxy Chain	N	N	N	N	N	N	any	B
Outbound Proxy	Y	N	B	N	B	N	any	B
Inbound Proxy	N	Y	N	B	N	B	any	B
Local Proxy	Y	Y	B	B	B	B	any	B

Given the four connection management mode characterization, we can obtain the whole test scenario space by enumerating all the configuration variables. To better understand the possible test cases, we show a unified logical component graph of the testbed in Figure 4. The proxy server in the middle represents the server under test. Its function is logically split to a UAS-like component (UAS_L), which interacts with the UAC in the left (UAC_L), and a UAC-like component (UAC_R) which interacts with the UAS in right side (UAS_R). 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 UAC and UAS, respectively.

In proxy chain and local proxy modes, the UAC_L and UAS_R represent the actual UAC and UAS. In outbound proxy mode, the UAC_L represents the actual UAC but the UAS_R represents the UAS-like component of the incoming proxy server that is connected to the proxy server under test. In inbound proxy mode, the UAC_L represents the UAC-like component of the outgoing proxy server that is connected to the proxy server under test, and the UAS_R represents the actual UAS.

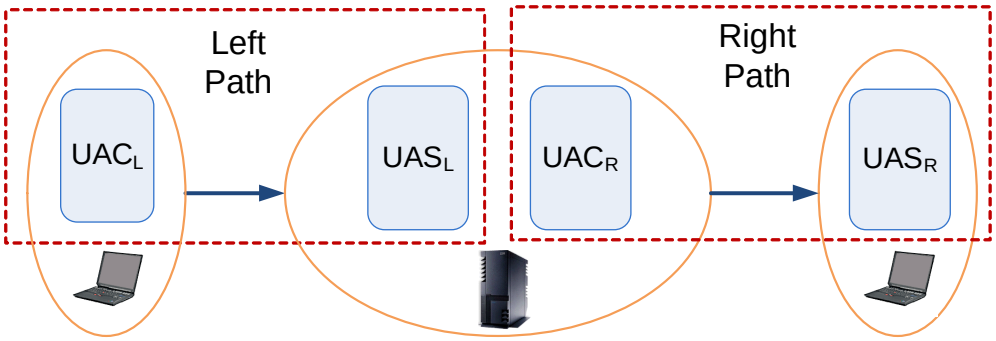


Fig. 4. Logical component graph of the SIP testbed

Different number of configuration variables is available for different scenarios. When UDP is used as the transport protocol, the only configuration variable we concern about is whether SIP proxy authentication is enabled. The cases with TCP and TLS transport are much more complicated because of the connection management possibilities as discussed in Section II-C. TCP and TLS tests may use single connection or multiple connection mode, and may have SIP proxy authentication enabled or disabled. If TLS is used, there are also the TLS session reuse, TLS mutual authentication, TLS cipher suite as configuration variables. In Table I we list the mapping between the configuration options and the four proxy connection modes for the TCP and TLS cases.

The “Left” and “Right” in the table represents the “Left Path” and “Right Path” as in Figure 4. The content entry value “N” means “No”, value “Y” means “Yes”, and B means both “Yes” and “No” are applicable. The only options applicable for TCP in Table I are the multiple connections option and the SIP proxy authentication option. For TLS, we can summarize three simple configuration rules for all scenarios from Table I: first, when a path (either the left or the right one) is in single connection mode (meaning value “N” for the “Multiple Connections” entry), connection handshake specific options do not apply. When a path (either the left or the right one) is in multiple connections mode (meaning value “Y” for the “Multiple Connections” entry), all connection handshake specific options apply. Second, the TLS cipher suite options apply to all scenarios. Third, the SIP proxy authentication options apply to left path multiple connections mode only (corresponding to multiple UAs connecting to their first hop SIP proxy server).

Expanding the whole test space in Table I 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 space. First, for TLS cipher suite, since the SIP standard [34] already specifies the mandatory TLS_RSA_WITH_AES_128_CBC_SHA cipher suite and a recommended TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher suite, we focus on these two cipher suites only. In particular, since the impact differences of 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 the mandatory TLS_RSA_WITH_AES_128_CBC_SHA only. Second, we enable SIP proxy authentication only in the outbound proxy and local proxy scenario, which is a common setting. Third, we test the TLS session reuse and TLS mutual authentication separately to understand each of their impacts. 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 to 16 test configurations for TCP and TLS. Adding the two UDP configurations, we have a total of 18 test configurations.

B. OpenSIPS SIP Server

The SIP server we evaluated is Open SIP Server (OpenSIPS) version 1.4.2 [25], 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)[14]. 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. In configurations involving proxy authentication where a user database is required, we use MySQL-5.0.67 [24], which we populated with $10,000$ unique user names and passwords.

We made several modifications to OpenSIPS in order to support all of our identified test cases in Section III-A. 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 options to OpenSIPS to request TLS session reuse when acting as a TLS client, and to request for TLS mutual authentication when it is acting as a TLS server.

The OpenSIPS "maximum number of TCP connections" parameter limits the number of TCP connections the server can handle. This value must be large enough so that new incoming TCP connections will not be dropped due to TCP connection number overflow. Although the server default value of $2,048$ is sufficient in most of our test cases, in high load test cases involving multiple connections between the proxy server and the UAS, we need to increase the value to $8,192$. We also increased the sizes of the TCP connection hash tables in OpenSIPS (TCP_ID_HASH_SIZE and TCP_ALIAS_HASH_SIZE) from $1,024$ to $2,048$. 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 scenario the peak throughput with SIP proxy authentication increased by close to an order of magnitude.

OpenSIPS has a children parameter that specifies the number of child processes that should be forked to simultaneously handle signaling messages. We started our evaluation with the default setting of children=4. When we compare the results with another setting children=1, we found that surprisingly the latter consistently performs equally well or better than the former. A detailed profile and Cycles Per Instruction (CPI) analysis on selected scenarios reveals that in the TCP case, although the CPI is indeed higher in the children=1 case, the number of instructions is significantly higher in the children=4 case, particularly in the kernel, so much so that it is more significant than the CPI. This suggests context-switching overhead in terms of code paths is the dominant cause leading to the difference. The trends still hold in the TLS case, although less pronounced. For this reason, we set children=1 in our test evaluations below.

C. SIPp Client Load Generator

We use another freely available open-source tool, SIPp [13] 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). SIPp is also extensible by writing third-party XML scripts that define new call flows; we wrote new flows that were not included with SIPp to handle authentication. SIPp has many run-time options we took advantage of, such as multiple transport (UDP/TCP/TLS) support; MD5-based hash digest authentication, and scriptable support to allow calls to be generated from a list of users. 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 options to SIPp to request TLS session reuse when acting as the TLS client and to request TLS mutual authentication when acting as the TLS server. SIPp has a max_socket option which sets the maximum allowed number of simultaneously open sockets. In test cases where each SIP call will create a new connection, we set max_socket=65535. The TLS support library for SIPp is a statically-compiled version based on OpenSSL [26] release 0.9.8i (which is the latest release at the time of the compilation).

D. 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. 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.

E. Operating System Software

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. To support large numbers of TCP connections, we modified the runtime Linux kernel configurations via the sysctl command to increase the number of available ephemeral ports to $55,000$ and to increase the maximum number of open file descriptors to $1,000,000$. 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:SSLroutines:SSL3_WRITE_PENDING:badwriteretry. A bug fix is found at [12]. 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.

F. 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 in calls per second as reported by SIPp and server profile CPU events as reported by oprofile. 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.

IV. Results and Analysis

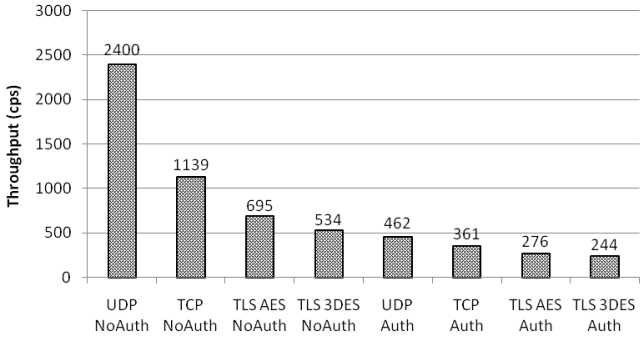


Fig. 5. Peak Throughput: Proxy Chain

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

A. Proxy Chain

Figure 5 shows the peak throughput in calls per second (cps) for the proxy chain scenario 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 TCP only, TLS with the TLS_RSA_WITH_AES_128_CBC_SHA cipher suite (abbreviated as TLS-AES), and TLS with the TLS_RSA_WITH_3DES_EDE_CBC_SHA cipher suite (abbreviated as TLS-3DES). Recall that in this scenario, 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 number of client machines to fully load the testbed. We take this factor into account in our cost model analysis in Section V by scaling by CPU utilization appropriately.

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 authentication 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.

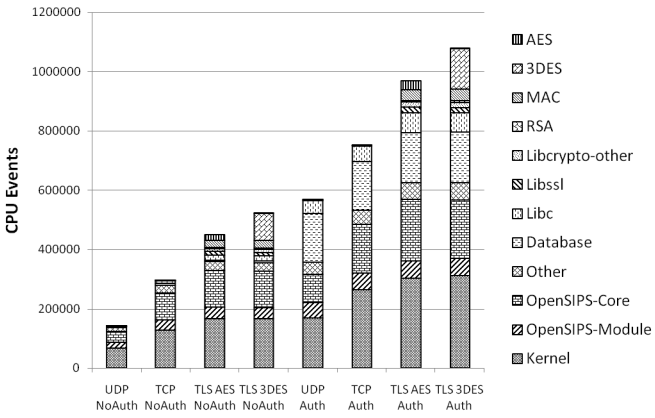


Fig. 6. CPU Profile Cycle Costs: Proxy Chain (50 cps)

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 ones extrapolated based on the CPU cycle costs observed. On average, the estimates match the observed peaks within 15 percent . Those are presented in Section V-B.

Figure 6 shows the number of non-idle CPU cycles consumed by the server in the proxy chain scenario for each configuration during the duration of the test. The mapping between the oprofile reported functions and the categories shown in the figure is listed in Appendix VII. We see that the total cost of the baseline UDP case without SIP authentication is about 144 thousand 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. Interestingly, AES itself only adds about 19K cycles in cost; 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. 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.

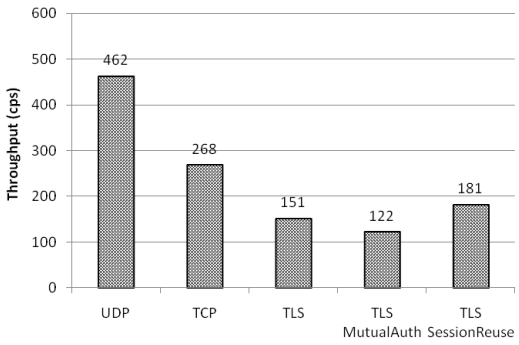


Fig. 7. Peak Throughput: Outbound Proxy

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 the 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.

B. Outbound Proxy

Figure 7 shows the peak throughputs of the outbound proxy scenario for several configurations. Recall that in the TCP or TLS cases of this scenario, each call results in a new connection being established with the server, as opposed to the proxy chain scenario above. Each bar again indicates a different configuration, namely UDP, TCP (and no TLS); 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 IV-A, for simplicity we restrict our experiments with TLS to use only AES for the remainder of this paper. The average CPU utilization

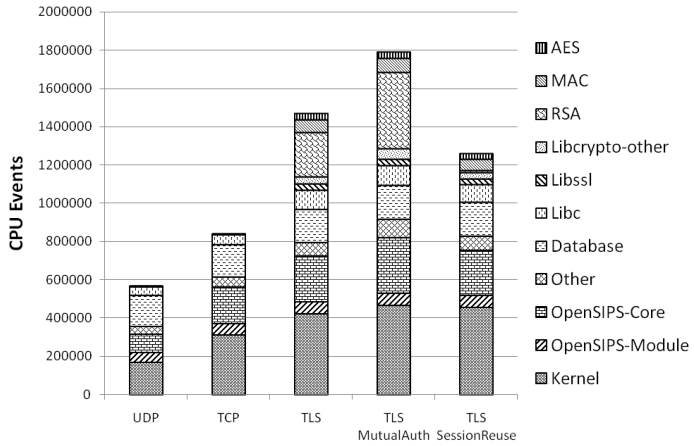


Fig. 8. CPU Profile Cycle Costs: Outbound Proxy (50 cps)

in each case is around 90%. We see that the peak throughput in 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%. To explain these differences we again turn to the profiles.

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 certification 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 socket layer crossings and network packets transmitted.

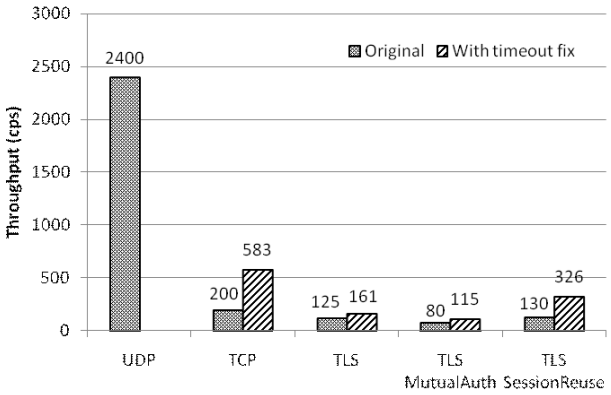


Fig. 9. Peak Throughput: Inbound Proxy

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. Since SIP authentication is enabled, we see database cost of about 29% in UDP and 20% in TCP and $10-15\%$ in the TLS case.

It is worth noting that the TLS mutual authentication test above includes SIP proxy authentication at the same time. One might argue that SIP proxy authentication may not be necessary with TLS mutual authentication where the server authenticates the client anyway. The point here is that the outbound proxy mode commonly requires not only user authentication, but also user authorization. The cost of SIP proxy authentication above is mainly attributed to the database operation, which is indeed for user authorization that is necessary even when TLS mutual authentication is used.

C. Inbound Proxy

Figure 9 shows the peak throughput of the inbound proxy scenarios. The configurations are the same as those in Figure 7 in Section IV-B, except that SIP 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. 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 is 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 relatively smaller impact. For the remainder of this Section we focus exclusively on OpenSIPS results where the timeout fix has been applied.

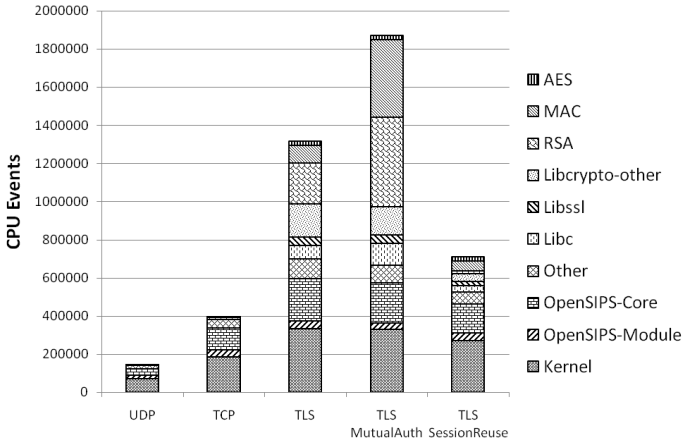


Fig. 10. CPU Profile Cycle Costs: Inbound Proxy (with Timeout Fix)

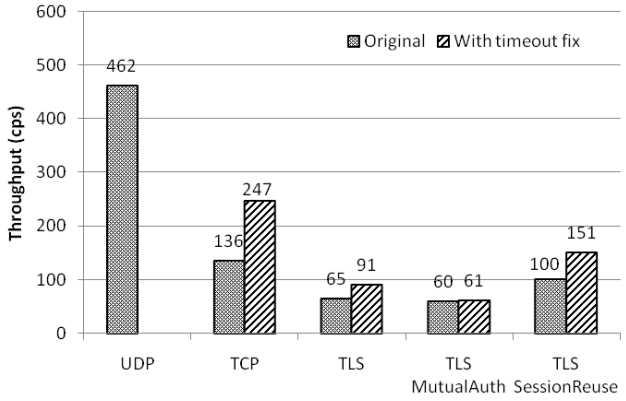


Fig. 11. Peak Throughput: Local Proxy

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%. We again turn to the profiles for explanation.

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' 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 needs to verify the certificates presented by the UAS, which was not present in earlier cases. In addition, in the mutual TLS case, the 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

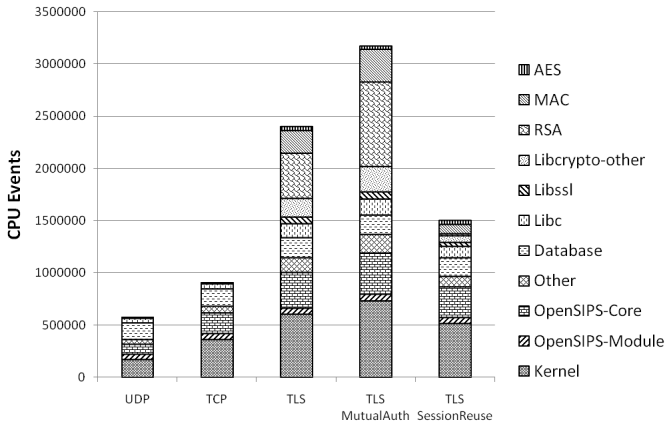


Fig. 12. CPU Profile Cycle Costs: Local Proxy (with Timeout Fix)

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.

D. Local Proxy

Figure 11 shows the peak throughputs of various configurations in the local proxy scenario, both with and without the timeout fix described in Section IV-C. Configurations are the same as in Figure 7; SIP authentication is 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. 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 33%, while enabling session reuse increases throughput 66%.

Figure 12 shows the CPU profile results for the local proxy scenario 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, 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, 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 IV-C.

V. A Component Cost Model

In this section we present a component cost model to help understand where the overheads in deploying SIP over TLS are and to aid network administrators in provisioning their systems. While clearly performance will vary across systems, our model helps provide guidance on relative performance across a single system. Thus, if an administrator understands how much server resources are required to support a SIP subscriber base using UDP, the cost model helps them estimate the capacity relative to that required to support TLS.

A. Constructing the Component Cost Model

Our model is based on decomposing the costs from each scenario into basic building blocks. Costs are derived from the number of CPU events, as measured by oprofile, that a particular proxy scenario configuration incurs at a load of 50 cps as

described in Section IV. We start from the most simple baseline configuration, proxy chain with UDP, and build up from there, normalizing that cost as one unit. For example, the baseline proxy chain mode with UDP does not include per-call connection establishment; this is a cost that will be calculated later. Next, we calculate the incremental overhead of TCP data transfer by subtracting the CPU events cost for the UDP proxy chain mode from the cost from the TCP proxy chain. Similarly, the TLS case in the proxy chain mode adds TLS bulk data encryption overhead to the plain TCP case. By subtracting the cost in the plain TCP case from the TLS case, we can obtain the cost of bulk data encryption. As long as the same cipher suite is used, this cost of bulk data encryption should be the same in all other scenarios. Next, if we look at the inbound proxy mode, the cost difference between the plain TCP inbound proxy mode and the plain TCP proxy chain mode is caused by the per-call TCP handshake overheads. Subtracting these two, we can calculate the normalized per-call handshake cost, which would be applicable also in the TLS inbound proxy mode. Following this approach, we can obtain the modular costs of all other components for the proxy chain, inbound proxy and outbound proxy scenarios. These costs are plotted in Figure 13.

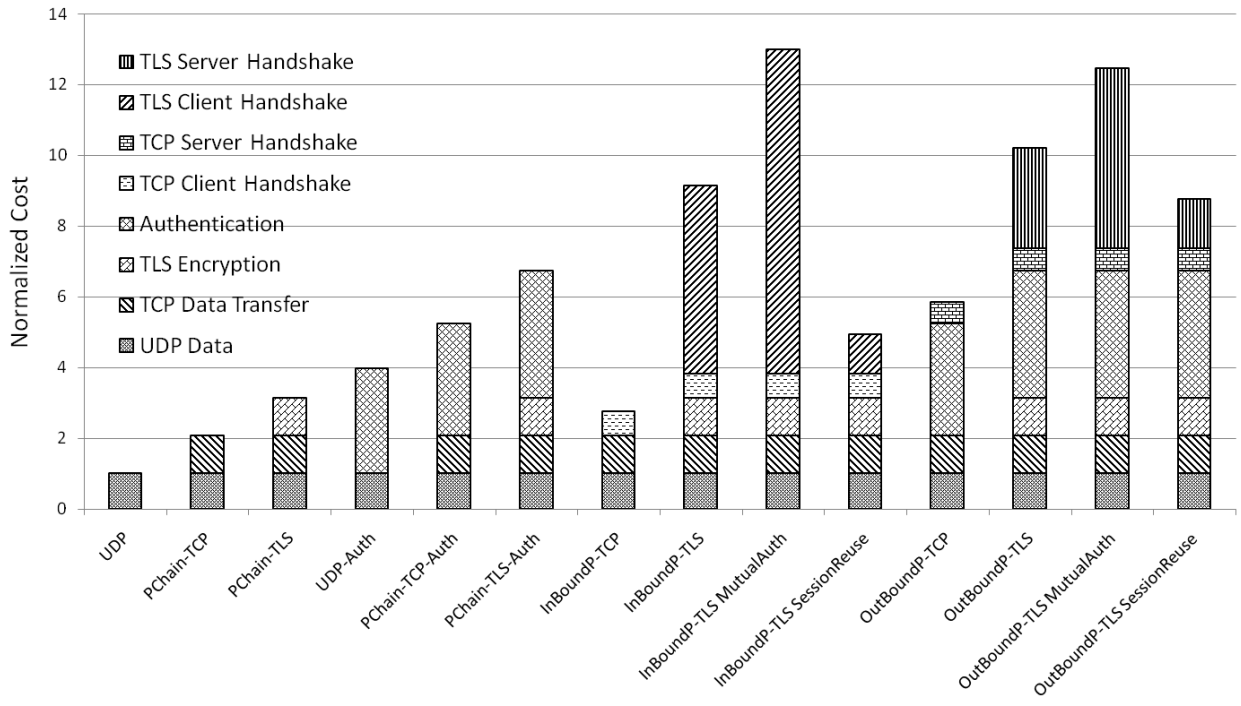


Fig. 13. Functional Components Cost Model

Figure 13 gives a simple model to compute the SIP proxy costs under varying settings. Now we explain in more detail each of these functional components and compare them in difference scenarios.

The UDP Data Transfer cost represents the base processing cost over UDP transport. Its main components are OpenSIPS-Core, OpenSIPS-Module, related kernel and libc costs. These costs are the minimum costs that will incur in any other scenarios. Therefore, it is used as the base for our cost normalization.

The TCP Data Transfer cost stands for the additional processing cost incurred when TCP is used instead of UDP. This cost is 1.1, a little larger than the base UDP cost. Using TCP thus more than doubles the cost of SIP processing with UDP.

TLS Encryption cost is the cost for bulk data encryption and decryption in any scenario involving TLS. This cost is determined by the encryption/decryption algorithm in the TLS cipher suite. In the majority of our tests, we used the AES cipher suite that the SIP RFC mandates with a 128-bit key size. The normalized cost of bulk data encryption using AES is 1.1, representing a similar amount of cost increase as the additional TCP data transfer cost. Adding bulk data encryption and TCP thus triples the cost of UDP with non-encrypted data. In section IV-A, we have seen that using the 3DES cipher instead of AES is 50% more costly, resulting in a 250% cost increase over the baseline UDP case.

The Authentication cost represents the cost of the SIP proxy MD5-based digest authentication mechanism. The values are 3 for UDP, 3.2 for TCP and 3.6 for TLS, respectively, which are over three times the base UDP data transfer cost. The authentication cost over TLS is more expensive than the cost of TCP due to additional TLS overheads. The sheer majority of the authentication cost is contributed by database lookup for credential verification. It should be possible to significantly reduce the database cost by replacing it with an in memory database.

The TCP Client Handshake cost represents the overhead when the proxy needs to open a TCP connection to the next hop on a per-call basis, as is the case in the inbound and local proxy modes. Similarly, the TCP Server Handshake cost represents the cost when the proxy must accept and establish a new TCP connection from the previous hop. Our experiments show that the costs at the TCP client and server side are similar, at between 0.6 and 0.7 of the base UDP transfer cost.

The TLS Client Handshake cost represents the overhead when the proxy needs to open a TLS session for a call, such as in the inbound and local proxy modes. The TLS Server Handshake cost represents the overhead when the proxy needs to accept a TLS session, as in the outbound and local proxy modes. The actual overheads depend on how TLS operates. With the normal TLS handshake, the cost at the client side and server side are 5.3 and 2.8 respectively. When TLS mutual authentication is enabled, the cost at the client and server side nearly doubles at 9.2 and 5.1 respectively. With TLS session reuse, the TLS client side cost reduces by 80% to 1.1 and the TLS server side cost shrinks by 50% to 1.4. A surprising observation is that the TLS client side cost is actually much higher than the TLS server side cost in both the normal TLS and TLS mutual authentication scenarios, which is contrary to the common wisdom [30]. We investigated this problem and identified a performance fix which significantly cuts the client side cost in the TLS, TLS mutual authentication and TLS session reuse cases by 50%, 55% and 73%, respectively. The fix is described in more detail in Appendix D.

B. Validating the Component Cost Model

Our component cost model is derived at a particular load point of 50 cps. Its applicability on higher load values requires the assumption that the CPU costs scale linearly as load increases. To test this assumption and verify the model, we took two steps.

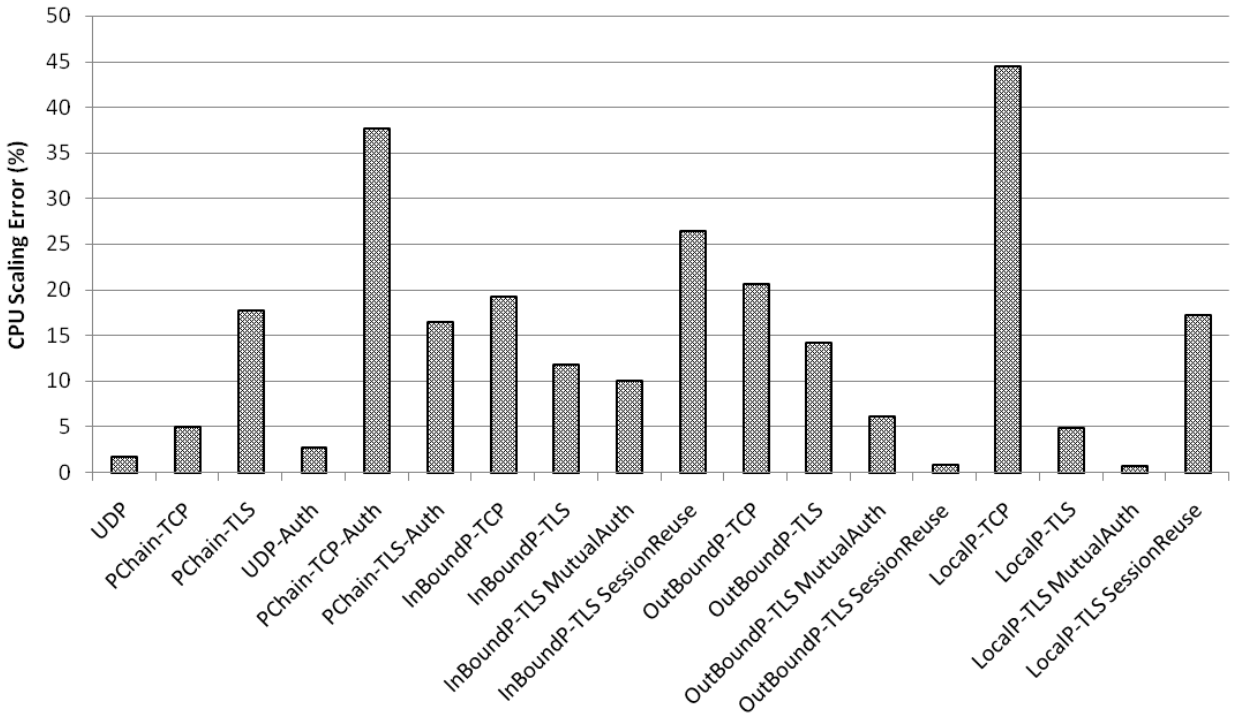


Fig. 14. CPU Scaling Error within Each Proxy Configuration

The first step is to verify that, within a particular proxy scenario configuration (e.g., proxy chain with TLS and session reuse), the peak throughputs are close to what we would "expect" them to be. In other words, given a throughput of 50 cps for some configuration, we estimate the peak throughput to be a linear extrapolation based on the CPU utilization at the 50 cps load level. For example, if for a particular configuration, we see 10% CPU at 50 cps, we expect the peak throughput to be close to 500 cps. Since different peak throughputs exhibit different CPU utilizations, we scale the estimates based on the utilization seen at the peak. We calculate the percentage error between the extrapolated estimate and the actual observed peak throughput in Figure 14. Although there are a few cases where the difference is up to 35% to 45% percent, the majority of the scenarios have much smaller errors. The overall average error is less than 15% percent. This indicates the CPU scaling assumption is reasonably effective.

The second step to verify the model is to check that the relationship between CPU events and CPU utilization is also linear. The reason is that we wish to use the cost model to predict peak throughput relationships across different scenarios. Our experience is that the number of CPU events is more stable estimate than CPU utilization, which has higher variability, particularly at low loads. If the event cost and CPU utilization across different scenarios exhibits a linear relationship, and since we saw above that throughput scales linearly with CPU utilization, we can similarly scale the event cost within each scenario. This lets us obtain a predicated peak throughput relationship across different scenarios by taking the inverse of the cost for each scenario at the 50 cps load level. Figure 15 shows the number of CPU events measured vs. CPU utilization across all 18 of our peak throughput measurements. The Y-axis presents the CPU event cost as measured by oprofile. The X-axis is the corresponding CPU utilization for that experiment. We also plot a fitted trend line, which shows a clear linear relationship. There are a few outlier points which are relatively farther away from the trend line, and as was expected, these are exactly the points which have the largest CPU utilization scaling error in Figure 14.

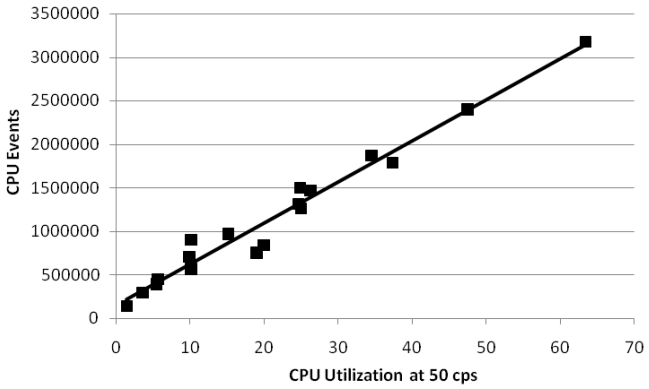


Fig. 15. CPU Events vs. CPU Utilization Across Different Proxy Configurations

C. Using the Component Cost Model

The component cost model can be used in at least two ways. First, given the component costs of a baseline scenario on a target system, the model offers a simple approach to approximate the relative cost of the SIP server operating at different modes. For example, the local proxy mode can be considered as a combination of the inbound proxy and outbound proxy mode. Given the costs of the inbound and outbound proxies, we can then derive the projected cost of the local proxy mode from this model. Figure 16 compares the model derived costs and the actual measured costs in the local proxy mode. We found the difference was between 3% to 13%, indicating a close match. Similarly, if we choose to use a different bulk data encryption algorithm in any of the scenario, we can substitute the cost of the encryption component with that of the new algorithm and keep the remainder the same.

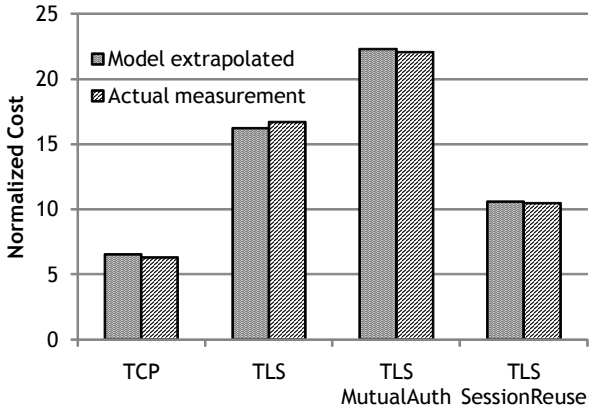


Fig. 16. Predicted vs. Actual Measured Cost in Local Proxy Mode

The second and more common use of the functional cost model is to approximate the peak throughput of different scenarios. Assuming that the cost (and CPU utilization) scales relatively linearly according to the load, as shown in Figure 14 in Section

V-B, the peak throughput should be inversely proportional to the cost. Therefore, if we know the peak throughput of the baseline UDP proxy chain scenario, we are able to project the peak throughput of other scenarios with different configuration combinations.

From Figure 13, we see that depending on whether authentication is enabled, the use of TCP reduces throughput by 51% or 81% over UDP in the proxy chain scenario. When the TCP connection handshake and maintenance costs are incurred as in other scenarios, the throughputs drop by 64% in the inbound mode and 83% in the outbound and local proxy mode compared to UDP.

When using TLS, if only bulk data encryption is used as in the proxy chain mode, the model suggests that TLS (with AES) reduces throughput by 35% to 25% depending on whether SIP proxy authentication is enabled. When both bulk data encryption and TLS handshake costs are incurred in the other proxy modes, the use of TLS reduces throughput by 65% to 70% in the inbound proxy and local proxy modes and 43% in the outbound proxy mode. Within the TLS cases, TLS mutual authentication may reduce the throughput from 18% to 25% depending on the proxy mode. When TLS session reuse is enabled, the throughput is increased by 16% in the outbound proxy mode and by $50-70\%$ in the inbound proxy and local proxy modes. The reason the improvement is much more dramatic in the latter scenarios is that the baseline TLS throughput in these modes is much smaller than the baseline throughput in the outbound proxy mode. This is due to the higher cost when the SIP server is acting as a TLS client.

VI. Related Work

The performance cost of SSL/TLS has been studied by a number of researchers, however, almost all these studies are based on SSL/TLS Web servers. Apostolopoulos et al. [3] 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. [15] investigated the architectural impact of SSL, and concluded that the use of SSL increases the compositional cost of transactions by a factor of $5-7$. Zhao et al. [42] 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. The execution breakdown of the individual component costs vary along the request file size. The RSA public key operations could take up to 90% of the total SSL processing cost when the file size is small. Symmetric (private) key encryption is negligible for small file sizes, but can increase significantly as the file size become larger. Coarfa et al. [5] measured the difference of 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. Other TLS costs are balanced across the various cryptographic and protocol processing steps. They also determined that even if the RSA operation costs can be reduced to zero, there is still a big performance difference between a TLS Web server and a traditional non-secured Web server.

Zeng and Cherkaoui [41] 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 1001 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 [39], a suite of SIP benchmarks for measuring SIP server performance on common tasks. Cortes [6] measured the performance of four different stateful SIP proxy server implementations over UDP and reported throughput results from $90-700$ cps. Nahum et al. [18] 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 indicated that any of these configurations can affect performance by a factor of $2-4$. The SIP over TCP transport scenario in [11] is limited to the TCP single connection mode which corresponds to the proxy chain scenario in this paper. Oho and Schulzrinne [23] studied the performance of the SIPd [37] SIP server over UDP and TCP transport. Their TCP tests include the multiple connection mode between the SIP proxy and the UA similar to the local proxy scenario of this paper. The difference is that in [23] the UAS closes a TCP connection after each transaction while in our tests the proxy server is responsible for tearing down the connection after the SIP session is over. The reported throughput is around 900 cps for UDP and 700 cps for TCP. Ram et al. [27] provide 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.

Salsano et al. [35] measured the performance of a Java-based SIP proxy server over UDP, TCP and TLS. This is the only work we are aware of that explicitly reports SIP over TLS performance. However, their SIP over TLS test is fairly simplified: they only tested the single connection mode, and the peak throughput of the server is at the order of 10 cps, which may undermine the representativeness of the results.

VII. Conclusions

We have evaluated and analyzed the impact of using TLS as a transport on SIP server performance versus the standard approach of using SIP over UDP. Using an experimental testbed with the OpenSIPS server, OpenSSL, Linux, and an Intel-based

server, we show that performance 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 two and half times that with TLS (the outbound proxy scenario with TLS session reuse); in the worst case, UDP is nearly 20 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 7 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 by up to a factor of 3.

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. We provide a measurement-driven cost model for operators to use in provisioning SIP servers with TLS. Our cost model predicts performance within 15 percent on average.

REFERENCES

[1] SIP forum. http://www.sipforum.org.

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

[3] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *IEEE InfoCom*, New York, NY, March 1999.

[4] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitima, and D. Gurle. Session initiation protocol (SIP) extension for instant messaging. RFC 3428 (Standard), December 2002.

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

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

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

[8] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFC 3546.

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

[10] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[11] E. Nahum and J. Tracey and C. Wright. Evaluating SIP server performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 349–350, June 2007.

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

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

[14] IPTel.org. Sip express router (SER). http://www.iptel.org/ser.

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

[16] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[17] A. Keromytis. Voice over ip: Risks, threats and vulnerabilities. In *Proceedings of the Cyber Infrastructure Protection (CIP) Conference*, June 2009.

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

[19] NIST. Data encryption standard DES, December 1993. http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[20] NIST. Digital signature standard DSS, May 1994. http://www.itl.nist.gov/fipspubs/fip186.htm.

[21] NIST. Secure hash standard, federal information processing standards publication 180-1, April 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[22] NIST. Advanced encryption standard (AES), federal information processing standards publication 197, November 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[23] K. Ono and H. Schulzrinne. One server per city: Using TCP for very large SIP servers. In *LNCS: Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, volume 5310/2008, pages 133–148, Oct 2008.

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

[25] The OpenSIPS Project. The open SIP server (OpenSIPS). http://www.opensips.org.

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

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

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

[29] E. Rescorla. openssl-examples. http://www.rtfm.com/openssl-examples.

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

[31] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006.

[32] R. Rivest. The MD5 Message-Digest Algorithm . RFC 1321 (Informational), April 1992.

[33] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[34] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320.

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

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

[37] H. Schulzrinne. SIPd. http://www.cs.columbia.edu/IRT/cinema.
[38] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.
[39] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. Sipstone-benchmarking SIP server performance. April 2002. http://www.sipstone.com.
[40] X. Wang, R. Zhang, X. Yang, X. Jiang, and D. Wijesekera. Voice pharming attack and the trust of voip. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–11, New York, NY, USA, 2008. ACM.
[41] Y. Zeng and O. Cherkaoui. Performance study of COPS over TLS and IPsec secure session. In *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 133–144, London, UK, 2002. Springer-Verlag.
[42] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of SSL processing. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 197–206, March 2005.

APPENDIX A: SECURITY AND CRYPTOGRAPHY IN TLS

Many Internet applications have a pressing need for security. TLS [9] is one of the two major existing Internet security standards. The other one is Security Architecture for the Internet Protocol (IPSec) [16]. Both TLS and IPSec standards are transparent to applications. IPSec runs at the network layer and is most commonly used in architectures where a set of administrative domains or hosts share an existing trust relationship with one another, e.g., in virtual private networks. TLS occupies a protocol layer between the application and the transport (usually reliable transport such as TCP), and it is widely used among domains and hosts with no pre-existing trust association, for example, in secure web servers (HTTPS).

The most important security properties of Internet communications can be characterized by confidentiality, integrity and endpoint authentication. Data confidentiality protects the data from being viewed by unintended listeners; integrity ensures that the data received are exactly the same as the data sent; endpoint authentication guarantees that the communication party is indeed the one that it claims to be.

TLS uses three categories of cryptographic operations to achieve these desired security properties: symmetric key cryptography, asymmetric key cryptography and hashing.

Symmetric Key Cryptography: TLS ensures data transfer confidentiality by using symmetric key encryption techniques such as RC4 [36], DES and 3DES [19], and AES [22].

RC4 is the best known stream cipher encryption algorithm. In this algorithm, a function generates a cryptographically secure pseudorandom key stream one byte at a time. Each byte of the key stream is then combined (e.g. XOR) with a byte of plain text to get a byte of cipher text. The key length of RC4 can vary between 8 and 2048 bits. TLS always uses RC4 with a 128-bit key length. RC4 is very fast compared to other cipher algorithms.

DES, 3DES and AES are block cipher encryption algorithms. In these algorithms, data are encrypted in larger blocks using methods such as diffusion, substitution and transposition. A popular operation mode for block cipher algorithms is called Cipher Block Chaining (CBC). This mode creates a dependency between the encryption of each plain text block and the cipher text for the previous block of cipher text, therefore fixing a problem caused by parallelism across individual blocks of data. DES is by far the most widely used symmetric cipher and it uses a 64-bit block size with a relatively short 54-bit key space. 3DES is effectively running DES three times to make it more secure, but also about three times slower. AES has a minimum block size of 128 bits and three key length of 128, 192, and 256 bits. AES is newer and faster than DES.

Asymmetric Key Cryptography: The use of symmetric key encryption requires the communications parties to first acquire the symmetric (shared) key. TLS uses asymmetric key (public key) cryptography algorithms such as RSA [33] and DH[10]/DSS[20] to protect the exchange and agreement of the symmetric key.

RSA is the best known public key algorithm. In an RSA operation, each party has a pair of keys, one public key and one private key. The public key is made public and the private key is kept secret. When the public key is used to encrypt a message, only the party with the corresponding private key can decrypt the encrypted message and vice versa. In TLS, typically the client generates a secret and encrypts it using the public key of the server. The server can decrypt the secret because it is the only one that has the corresponding private key.

TLS also uses the reverse of the above operation to achieve authentication. For example, if the server wants to be authenticated by the client, the server encrypts a message with its own private key, which produces a digital signature. The client can then verify the signature by decrypting the message with the server's public key. The clients know the server is authentic because only the server has the corresponding private key that could have produced the encrypted message.

A remaining issue regarding RSA operation is how one party gets the other party's real public key. TLS uses X.509 certificates to associate a public key with an identity. The certificate is signed by a trusted Certificate Authority (CA). It is assumed that the party to authenticate the other party at least possesses the public key of the trusted CA. The CA signs the certificate using its private key. Therefore, the CA-signed certificate can be verified using the CA's public key. In the common web scenarios, the CA certificates are frequently bundled in the browser, thus users typically don't need to know or configure them.

DH (Dffie-Hellman) is a key agreement algorithm, which is different from a key exchange algorithm like RSA. With DH, each party combines its own private key with the other party's public key to collectively generate an agreed key that is private to both parties. DSS (Digital Signature Standard) is a digital signature only algorithm commonly used together with DH.

Message Digest: TLS provides message integrity through hash functions, also known as digest algorithms, such as MD5 [32] and SHA [21]. These algorithms take an arbitrary length message and output a fixed length digest string of the message. A good digest algorithm ensures that two different messages are unlikely to produce the same digest and it is extremely difficult to reversely compute a message given its digest. In TLS, the message digest algorithm is used to compute a keyed Message Authentication Code (MAC). The sending and receiving parties both compute the MAC and confirm message integrity by making sure the MAC computed by one party matches that computed by the other party.

APPENDIX B: SYSTEM CONFIGURATION FOR SCALABILITY TEST

We put the following contents in the `/etc/sysctl.conf` file to increase the system maximum number of file descriptors to 1,048,576, and the available port range from 10,000 to 65,535.

```
fs.file-max = 1048576
net.ipv4.ip_local_port_range = 10000 65535
```

We also edited the `/etc/security/limits.conf` file to increase the soft and hard limit of the number of open files for the login "user" to 1,000,000.

```
user soft nofile 1000000
user hard nofile 1000000
```

To allow a remote shell to access a large number of file descriptors via `ssh`, we use the `ulimit` command. For example, `ulimit -n 1000000` increases the number of file descriptors available to the shell to 1,000,000. On our Ubuntu 8.04 platform, we found this `ulimit` command over ssh can only be successfully executed by the "root" account. If a "sudo" user is to perform a remote `ssh` and try to get a shell with expanded number of file descriptors, we need to edit the `/etc/init.d/ssh` file to add a line `ulimit -n 1000000`, and restart `ssh` by `/etc/init.d/ssh restart`.

APPENDIX C: OPROFILE FUNCTION MAP DEFINITION

This section lists how we map function names obtained from oprofile results to the specific categories. The first level of classification is based on the application name shown in oprofile as in Table II.

To study the libcrypto category in more detail, we further classify the functions within the libcrypto library into AES, 3DES, MAC and RSA as listed in Table III through Table V. All other functions in the libcrypto library are grouped into the libcrypto-other category.

TABLE II
FUNCTION NAME AND APPLICATION MAPPING

Category	*Application*
Libcrypto	libcrypto.so.0.9.8
Libssl	libssl.so.0.9.8
Libc	libc-2.7.so
Database	mysqld, db_mysql.so
OpenSIPS-Core	opensips
OpenSIPS-Module	maxfwd.so, registrar.so, rr.so, sl.so, textops.so, tm.so, uri.so, usrloc.so, uri_db.so
Kernel	vmlinux-debug-2.6.24-19-server
Other	all others

TABLE III
FUNCTIONS CLASSIFIED AS AES AND 3DES

AES	*3DES*
_x86_AES_encrypt	DES_set_key_unchecked
AES_cbc_encrypt	DES_encrypt2
AES_set_decrypt_key	DES_ede3_cbc_encrypt
AES_set_encrypt_key	des_ede_cbc_cipher
_x86_AES_decrypt	DES_encrypt3
aes_init_key	DES_decrypt3
aes_128_cbc_cipher	

TABLE IV
FUNCTIONS CLASSIFIED AS MAC

ENGINE_get_digest_engine	EVP_MD_CTX_cleanup
sha1_block_asm_host_order	EVP_MD_CTX_init
HMAC_Init_ex	EVP_sha1
MD5_Final	sha256_block
EVP_DigestUpdate	SHA1_Final
EVP_DigestFinal_ex	EVP_MD_CTX_set_flags
EVP_MD_CTX_copy_ex	SHA1_Init
MD5_Update	SHA1_Update
sha1_block_asm_data_order	HMAC_Update
EVP_MD_size	EVP_MD_block_size
EVP_MD_CTX_test_flags	HMAC_CTX_cleanup
EVP_DigestInit_ex	md5_block_asm_host_order
HMAC_Final	SHA256_Update
EVP_MD_CTX_clear_flags	HMAC_CTX_init
MD5_Init	

TABLE V
FUNCTIONS CLASSIFIED AS RSA

RSA_free	x509_object_cmp
x509_name_ex_i2d	RSA_size
bn_mul_comba8	BN_num_bits_word
d2i_PublicKey	RSA_eay_public_encrypt
ASN1_tag2bit	BN_hex2bn
BN_BLINDING_invert_ex	BN_mul
asn1_enc_save	X509_TRUST_get0
BN_BLINDING_get_thread_id	ASN1_item_ex_i2d
asn1_ex_i2c	BN_from_montgomery
ASN1_put_object	X509_VERIFY_PARAM_free
BN_CTX_new	asn1_item_combine_free
ASN1_item_ex_d2i	bn_cmp_part_words
RSA_eay_private_decrypt	BN_mod_inverse
BN_rand	BN_CTX_free
BN_clear	bnrand
X509_STORE_get_by_subject	ASN1_get_object
ASN1_template_new	X509_STORE_CTX_init
BN_set_word	bn_cmp_words
X509_get_ext_d2i	BN_num_bits
X509_OBJECT_idx_by_subject	BN_CTX_get
bn_mul_recursive	BN_sub
RSA_padding_check_PKCS1_type_2	bn_rand_range
BN_sqr	RSA_new_method
ASN1_OBJECT_free	BN_mod_mul
X509_VERIFY_PARAM_inherit	EVP_PKEY_size
OPENSSL_cleanse	X509_NAME_oneline
BN_rand_range	BN_bn2bin
rsa_get_blinding	BN_mod_exp
bn_expand_internal	bn_mul_words
ASN1_primitive_new	bn_sqr_comba8
BN_sub_word	BN_MONT_CTX_init
BN_nnmod	ASN1_primitive_free
BN_bin2bn	X509_VERIFY_PARAM_lookup
BN_CTX_end	BN_div
BN_CTX_start	ASN1_item_free
BN_clear_free	i2c_ASN1_INTEGER
X509_OBJECT_retrieve_by_subject	ASN1_item_i2d
X509_verify_cert	RSA_private_decrypt
asn1_template_ex_d2i	BN_mod_exp_mont
asn1_template_ex_i2d	asn1_do_adb
BN_mod_exp_mont_consttime	BN_is_bit_set
BN_rshift	ASN1_object_size
BN_copy	BN_rshift1
X509_get_subject_name	bn_sqr_recursive
X509_free	EVP_PKEY_free
asn1_enc_restore	bn_mul_add_words
X509_STORE_CTX_cleanup	asn1_i2d_ex_primitive
X509_VERIFY_PARAM_new	BN_BLINDING_create_param
i2d_X509	i2c_ASN1_BIT_STRING
BN_usub	bn_add_words
RSA_eay_mod_exp	ASN1_item_verify
bn_sub_part_words	d2i_PrivateKey
BN_mod_mul_montgomery	BN_MONT_CTX_new
BN_uadd	bn_mul_normal
asn1_do_lock	BN_free
BN_BLINDING_convert_ex	BN_free
X509_get_issuer_name	asn1_get_field_ptr
ASN1_STRING_free	BN_init
X509_NAME_cmp	asn1_ex_c2i
ASN1_STRING_set	asn1_template_noexp_d2i
X509_get_pubkey_parameters	asn1_primitive_clear
BN_ucmp	BN_MONT_CTX_set
asn1_item_ex_combine_new	ASN1_template_free
x509_cb	X509_OBJECT_up_ref_count
ASN1_STRING_type_new	bn_sub_words
BN_add	X509_subject_name_cmp
HMAC_Final	BN_lshift
BN_value_one	asn1_d2i_ex_primitive
BN_set_bit	pubkey_cb
bn_expand2	

APPENDIX D: TLS CLIENT SIDE CONNECTION IMPROVEMENT

By enabling the debugging log and examining the code, we found that when OpenSIPS tries to establish an outgoing TLS connection as the client side, it calls the SSL_connect function, which returns SSL_ERROR_WANT_READ, indicating the function should be called when data can be read. However on the epoll based waiting loop, both POLLIN and POLLOUT events are being monitored. When either event is reported, the SSL_connect function is called again. During each connection establishment, there are a large number of times where the epoll loop returns POLLOUT which indicates data can be written but not read. All these returns incur additional calling of the SSL_connect function, which simply returns SSL_ERROR_WANT_READ again. These additional unnecessary calls to SSL_ERROR_WANT_READ (e.g., it could be 44 times for one connection setup) turn out to have a big cost impact. We modified the epoll loop so that during the connection setup, the loop only reports the needed events. Figure 17 shows the cost comparison with and without the fix in the inbound proxy TLS mode. The costs shrink in virtually all categories except AES encryption cost, which should not be affected. The total libcrypto cost is reduced by 38% from 500K down to 310K. The most significant cost reduction comes from libcrypto-other functions (128k, including 27k ssleay_rand_add, 11k int_thread_get, 10k int_thread_get_item, 6k int_thread_release, 7k lh_retrieve, 19k err_clear_error, 7k err_get_state, 7k crypto_add_lock, 4k crypto_free). MAC and RSA costs are also reduced by 34k and 28k, respectively. In addition, the overall cost is seen to be reduced by 29%, from 1,316K down to 936K. Cost savings from outside the libcrypo library include libssl (22k), libc (37k), Other (29k), OpenSIPS-Core (39k), OpenSIPS-Module (4k) and Kernel (58k).

To further verify the TLS libcrypto cost associated with establishing TLS connections (as TLS client) and accepting TLS connections (as TLS server), we compare the corresponding cost incurred in our SIP proxy with that incurred in a simple HTTPS client server application [29], assuming a similar number of connections are being set up. Results show that the TLS client side cost in the two cases are around 310k and 270k, respectively, with an RSA cost of around 150k in both cases. The TLS server side cost in the two cases are 340K and 400K, with RSA cost of 240k and 300K, respectively. These results indicate a reasonably close crypto costs match between the relatively complex SIP proxy server and the simple HTTPS server.

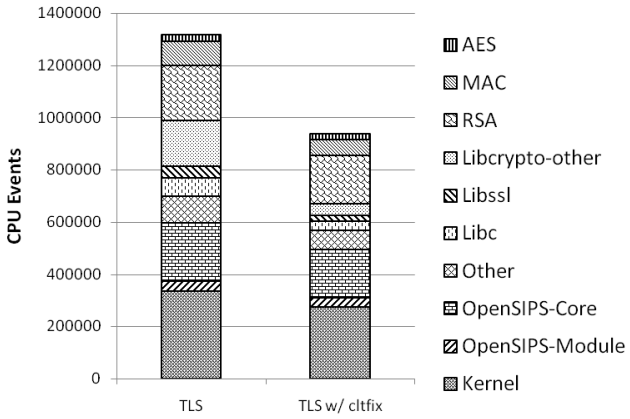


Fig. 17. Impact of TLS Client Side Fix on CPU Events in Inbound Proxy TLS Mode

We update the component cost model with the TLS client fix applied in Figure 18. As we can see, the client side cost in the TLS, TLS mutual authentication and TLS session reuse cases is reduced by 50%, 55% and 73%, respectively. The corresponding total cost savings are 29%, 38%, and 17%, which represent a potential peak throughput increase of 41%, 62%, and 21%.

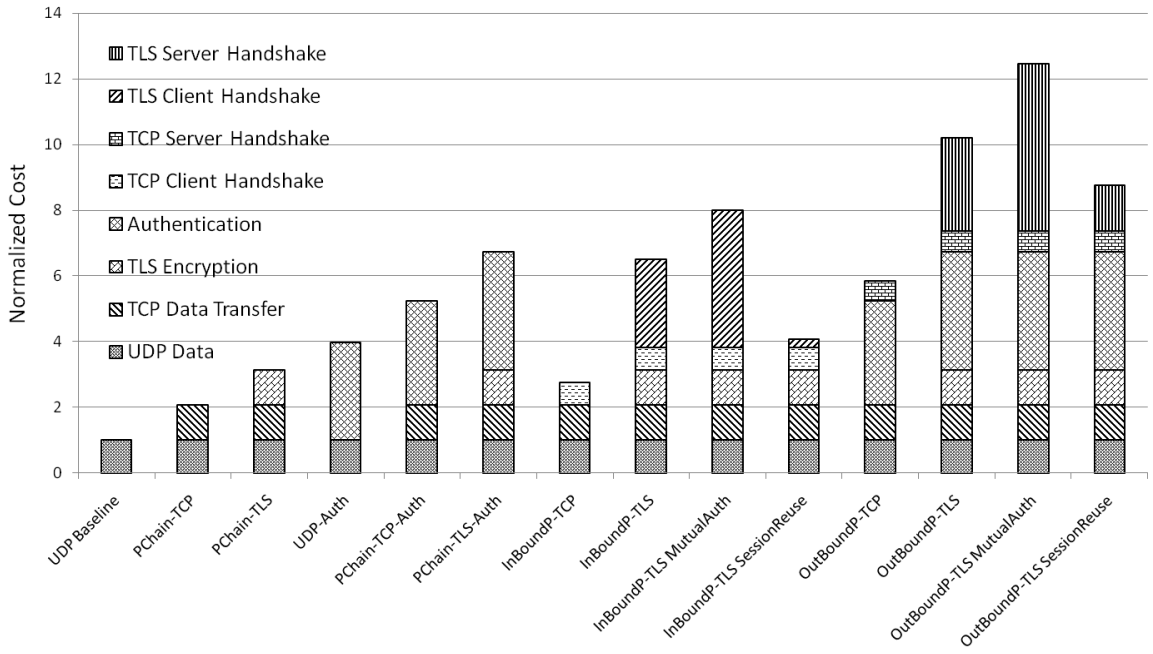


Fig. 18. Component Cost Model with TLS Client Fix Applied