# The Impact of TLS on SIP Server Performance: Measurement and Modelling

Charles Shen, Erich Nahum, Henning Schulzrinne, and Charles P. Wright

***Abstract*—Securing VoIP is a crucial requirement for its successful adoption. A key component of this is securing the signaling path, which is performed by SIP. Securing SIP is accomplished by using TLS instead of UDP as the transport protocol. However, using TLS for SIP is not yet widespread, perhaps due to concerns about the performance overhead.**

**This paper studies the performance impact of using TLS as a transport protocol for SIP servers. We evaluate the cost of TLS experimentally using a testbed with OpenSIPS, OpenSSL, and Linux running on an Intel-based server. We analyze TLS costs using application, library, and kernel profiling, and use the profiles to illustrate when and how different costs are incurred, such as bulk data encryption, public key encryption, private key decryption, and MAC-based verification.**

**We show that using TLS can reduce performance by up to a factor of 17 compared to the typical case of SIP-over-UDP. The primary factor in determining performance is whether and how TLS connection establishment is performed, due to the heavy costs of RSA operations used for session negotiation. This depends both on how the SIP proxy is deployed (e.g., as an inbound or outbound proxy) and what TLS options are used (e.g., mutual authentication, session resumption). The cost of symmetric key operations such as AES, 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.**

***Index Terms*—Computer networks, Security, Internet telephony, Performance evaluation**

## I. Introduction

Securing Voice over IP (VoIP) is a necessary requirement for enabling its stable, long-term adoption. A key aspect of VoIP security is securing the signaling path, typically provided by the Session Initiation Protocol (SIP) [49]. SIP is an application layer signaling protocol for creating, modifying, and terminating media sessions in the Internet. Major standards bodies including 3GPP, ITU-T, and ETSI have all adopted SIP as the core signaling protocol for services such as VoIP, conferencing, Video on Demand (VoD), presence, and Instant Messaging (IM). Like other Internet services, SIP-based services may be susceptible to a wide variety of security threats including social threats, traffic attacks, denial of services and service abuse [7], [12], [34]. One of the main reasons that enable these threats is the common use of clear text SIP signaling over any transport that is susceptible to eavesdropping and replay attacks, 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 [13]). It has also been shown that even commercial VoIP services may be prone to large-scale voice pharming [60], where victims are directed to fake interactive voice response systems or human representatives for revealing sensitive information.

Transport Layer Security (TLS) [20], based on the earlier Secure Sockets Layer (SSL) [25] specification, is a widely used Internet security protocol occupying a layer between the application and the transport layer. SIP specification [49] lists TLS as a standard method to secure SIP signaling. Various other organizations and industrial consortiums have also recommended the use of TLS for SIP signaling. For example, the SIP Forum [6] mandates TLS for interconnecting enterprise and service provider SIP networks in its specification document.

However, while interest in securing SIP is growing [43], [59], actual large scale deployment of SIP-over-TLS has not yet occurred. One important reason is the common perception that running an application over TLS is costly compared to running it directly over TCP or UDP. 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. TLS works over both UDP (Datagram TLS [46]) and TCP, we focus our study on using TLS over TCP because it is used by the majority of TLS implementations today.

This paper makes the following contributions:

- We present an experimental performance study of the impact of using TLS on SIP servers. Our study is conducted using Open SIP Server (OpenSIPS) [41], which is one of the de facto open source version of SIP servers, occupying a role similar to that of Apache for web server [9], [11], [18], [19], [21], [23], [36], [42], [61]. We use the OpenSSL [4] library in Linux on an Intel-based server and evaluate the CPU cost of TLS under four SIP proxy usage modes: proxy chain, outbound proxy, inbound proxy, and local proxy. We show that using TLS can reduce performance by up to a factor of 17 compared to the typical case of SIP-over-UDP.

Manuscript received Feb. 14, 2011; accepted Nov. 3, 2011; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Associate Editor Prof. Z Morley Mao.

Charles Shen is with AT&T Security Research Center, 33 Thomas St., New York, NY, 10007 USA (email: see http://www.charleshen.com).

Erich Nahum and Charles P. Wright are with IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532 USA (email: {nahum,cpwright}@us.ibm.com).

Henning Schulzrinne is with Department of Computer Science, Columbia University, 1214 Amsterdam Ave., New York, NY 10027 USA (email: hgs@cs.columbia.edu).

- We use application, library, and kernel profiles to examine, analyze, and explain performance differences. The profiles illustrate how costs are incurred under different scenarios and how they can be reduced, e.g., extra Rivest, Shamir and Adleman (RSA [48]) overheads observed when mutual authentication is used and disappeared when session resumption is performed. They also show some results that distinguish SIP server from other server scenarios, e.g., bulk crypto costs of Advanced Encryption Standard (AES [38]) or Triple Data Encryption Standard (3DES [37]) are small. In addition, the tests show how some overheads are due to mechanisms such as kernel overhead and SSL state management rather than simply crypto algorithms such as RSA or AES.
- We identify and solve several performance problems in OpenSIPS. Each is related to connection management with large numbers of connections under high loads. The fixes improve performance in some cases from a few times up to an order of magnitude.
- We provide a measurement-parameterized 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 [10], [15], [33], [63] or policy-based network management [62]. SIP protocol behavior is different from these protocols. SIP proxy servers can act as clients to other servers and therefore can incur large client-side TLS costs. SIP servers also have a much wider range of connection management behavior than other servers (see Section III-F), and this connection management is a 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 could be trivial in comparison.

The net result is that the performance cost of deploying SIP over TLS instead of directly over UDP can be significant, depending on how the SIP proxy server is used (e.g., as an inbound or outbound proxy) and how TLS is configured (e.g., with or without mutual authentication or session resumption). 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 testbed used and how we determine our test cases. Section IV presents our experimental results. Section V develops our cost model and Section VI describes related work.

## II. Background

### A. TLS Operation Overview

We provide a brief description of the TLS protocol. For more details, please see [20], [45], [51].

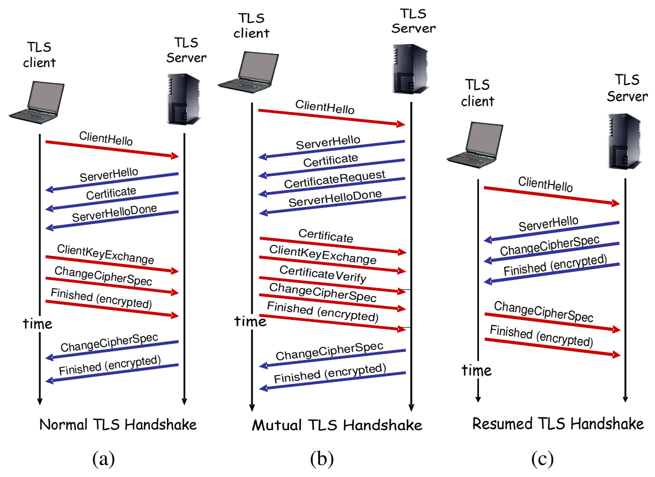

Fig. 1: TLS Handshake Message Flows

TLS operation consists of 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 secrets 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 a cipher suite indicating that RSA public key algorithm is used for shared secret key exchange and authentication; 128-bit AES in Cipher Blocking Chaining (CBC) mode is used for bulk data encryption; and SHA-1 [22] is used as the message digest algorithm to compute the Message Authentication Code (MAC).

A typical message flow in the TLS handshake phase is illustrated in Fig. 1a. 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 messages in this stage. Upon receiving the server's certificate, the client authenticates the server by verifying its certificate using the Certificate Authority (CA)'s public key. Depending on the key exchange mode, the client may then generate a `pre_master_secret`, and encrypt it using the server's public key obtained from the server's certificate. 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 will share based on the same pre_master_secret. The master_secret is further used to generate 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 messages that have been sent to the other party. The Finished message is used to ensure the integrity of the handshake.

In normal TLS handshake, only the client authenticates the server. TLS also offers a mutual authentication mode where the server can authenticate the client as well as shown in Fig. 1b. In this mode, sends an additional CertificateRequest message to request the client's certificate. The client responds with 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.

Since public key cryptography is usually more expensive than secret key cryptography, TLS uses public key cryptography to establish the shared secret key in the handshake phase, and then uses secret key cryptography during the bulk data encryption phase. To reduce costs during handshake, TLS also offers a session resumption mode as that allows two parties to avoid negotiating the pre_master_secret if it has been done previously within a time threshold. It is important to distinguish the notion of a *connection* versus a *session* in TLS. A TLS *connection* corresponds to one specific communication channel which is typically a TCP connection; while a TLS *session* is associated with a negotiated set of algorithms and the established master_secret based on the pre_master_secret. Multiple connections may be mapped to the same session, all sharing the same set of algorithms and the master_secret, but each with different symmetric keys for bulk data encryption. The notion of session resumption indicates the resumption of a previously negotiated set of cryptographic algorithms and the master_secret. The handshake message flow for TLS session resumption is shown in Fig. 1c. The session resumption 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).

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. We focus on stateful SIP proxying because many standard application functionalities, such as authentication, authorization, accounting all require the proxy server to keep different levels of session state information.

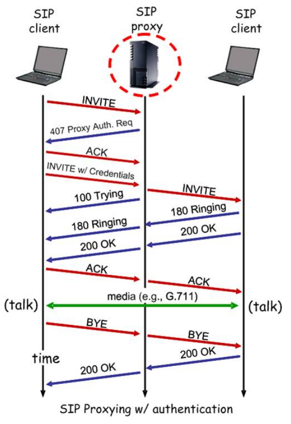

Fig. 2: SIP Stateful Proxying with Authentication

Fig. 2 shows a typical message flow of stateful SIP proxying. Two SIP UAs, designated as User Agent Client (UAC) and User Agent Server (UAS) represent the caller and callee of a multimedia session. The hashed circle around the proxy indicates that this is the server that we are measuring ("system under test"). In this example, the UAC wishes to establish a session with the UAS and sends an INVITE message to the proxy. The proxy server enforces proxy authentication by responding with a 407 Proxy Authentication Required message, challenging the UAC to provide credentials that verify its claimed identity (e.g., based on MD5 [47] digest algorithm). This proxy authentication step is optional but is usually deployed between a UA and its first hop SIP proxy server. The UAC then transmits a new INVITE message with the generated credentials in the Authorization header. After receiving and verifying the UAC credential, the proxy sends a 100 TRYING message to inform the UAC that the message has been received and that it needs not worry about hop-by-hop retransmissions. The proxy then looks up the contact address for the SIP URI of the UAS and, assuming it is available, forwards the message. The UAS, in turn, acknowledges receipt of the INVITE message with a 180 RINGING message and rings the callee's phone. When the callee actually picks up the phone, the UAS sends out a 200 OK. Both the 180 RINGING and 200 OK messages make their way back to the UAC through the proxy. The UAC then generates an ACK message for the 200 OK message. Having established the session, the two endpoints communicate directly, peer-to-peer, using a media protocol such as RTP [53]. 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. Note that although Fig. 2 shows a single SIP proxy server between the two UAs, in real networks it is common to have multiple proxy servers in the signaling path. The message flow with multiple proxy servers is similar, except that the proxy authentication is usually only applicable to the first hop.

### C. SIP Connection Management over TLS

Our SIP-over-TLS study is conducted on top of TCP transport since TCP is used by most of the TLS implementations today. In general, a TCP connection is first established between the endpoints, and then a TLS handshake occurs to negotiate the TLS session. Once the TLS session is established, the SIP signaling messages will be passed to the TLS layer and encrypted.

When a connection oriented transport such as TCP is used, the connection management policy needs to be defined. In a multi-hop SIP server network scenario, it is usually preferable to maintain a single long-lasting connection between two interconnected proxy servers [28]. Having all SIP messages between the two proxy servers 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 server typically has to establish separate connections with each of them since they are located on separate hosts.

## III. Testbed and Methodology

### A. OpenSIPS SIP Server

The SIP server we evaluated is OpenSIPS version 1.4.2 [41], a freely-available, open source SIP proxy server. OpenSIPS is a fork of OpenSER, which in turn is a fork of SIP Express Router (SER) [32]. All these proxy servers are written in the C language, considered to be highly efficient and represent the de facto open source version of SIP servers.

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

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

In configurations involving proxy authentication where a user database is required, we used MySQL-5.0.67 [2], which we populated with $10,000$ unique user names and passwords. The MySQL server runs on the same machine as the OpenSIPS server.

### B. SIPp Client Load Generator

For SIP traffic generator, we use the open-source SIPp [26] tool which allows a wide range of SIP scenarios to be tested. We also added additional functionality to SIPp to accommodate all our test cases, including the SIPp options to request TLS session resumption when it is acting as the TLS client and to request TLS mutual authentication when it is acting as the TLS server. The TLS support library for SIPp is a statically-compiled version based on OpenSSL [4] release 0.9.8i, which is the latest release at the time of the compilation.

### C. Hardware and Connectivity

Our server hardware has 2 Intel Xeon 3.06 GHz processors with 4 GB RAM and 34 GB disk drives. However, for our experiments, we only use one processor because SIP performance under multiple processors or a multi-core processor is itself a topic that requires separate attention [61]. 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. A longer round trip time should not have much impact on our evaluation as long as it does not exceed the 500 ms SIP-over-UDP retransmission threshold. Typical network round trip times are usually well below 500 ms [1] unless the connection includes satellite links, for example.

### D. Software Platform

The server uses Ubuntu 8.04 with Linux kernel 2.6.24-19, OpenSSL 0.9.8.g, and oprofile 0.9.3 [5]. The clients use Ubuntu with either a 2.6.22 kernel or a 2.6.24 kernel. We encountered an SSL library failure at the SIPp load generator side when generating high loads. After examining the OpenSSL error queue in more detail, we found the cause and the bug fix [24]. This fix was submitted in 2003 but had not 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.

### E. Workload and Performance Metrics

The workload is a standard SIP call flow the same as in Fig. 2. There is no call hold time. A call is completed when all messages (from the initial INVITE to the 200 OK in response to the BYE) are correctly delivered. Our main metrics include server throughput which counts the number of successfully completed calls per second (cps) as reported by SIPp, and server CPU events profile as reported by oprofile. We also

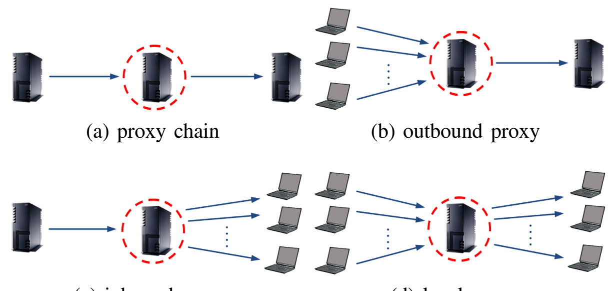

Fig. 3: SIP Proxy Operation Modes

measure server CPU utilization. Our test runs last for 120 seconds after a 30-second warm-up time. The results are the average of three consecutive test runs.

## F. *Test Matrix and Evaluated Test Cases*

We first group SIP server connection management configurations into four different deployment modes as shown in Fig. III-F.

1) Fig. 3a shows the *proxy chain* mode, where the proxy server interconnects two other proxy servers in a chain fashion. This is intended to model, e.g., how two core SIP proxy servers of different service providers communicate. Only one connection is needed for each neighboring proxy server in this case.
2) Fig. 3b shows the *outbound proxy* mode, where the proxy accepts multiple connections from UACs but only establishes a single outgoing connection with another proxy server. This configuration models how phones in an enterprise SIP deployment would make calls external to the organization.
3) Fig. 3c is the *inbound proxy* mode, where the proxy server under test accepts a single connection from an upstream proxy server and establishes multiple connections to individual UASes. This is the mirror of the outbound proxy configuration above, where incoming SIP traffic is routed to phones.
4) Fig. 3d, is the *local proxy* mode, where the proxy server under test connects UACs and UASes directly, and therefore accepts both incoming connections and creates outgoing connections simultaneously. This configuration is intended to model how phones in an enterprise deployment would communicate with each other.

The above four modes describe the full range of connection management behavior for SIP proxy servers, from completely persistent connections among proxy servers (the proxy chain mode) to non-persistent connections where each call requires a connection setup and teardown (the local proxy mode). In addition, the inbound and outbound cases distinguish where connections are passively accepted (the inbound proxy mode) vs. those that are actively created (the outbound proxy mode). Real proxy servers usually support all these operation modes, and operate in a mode somewhere in the middle of these four

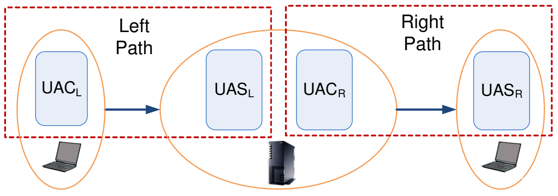

Fig. 4: Logical Component Graph of SIP Testbed

extremes. Thus this mode characterization is more logical than physical, but it helps us explore the design space fully.

To prepare the applicable test matrix, we list the five main configuration variables in our SIP-over-TLS tests along with the four operation modes in Table I.

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

Directly 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 numer of cases. First, we focus only on two TLS cipher suites, which are specified by the SIP standard [49] as mandatory (`TLS_RSA_WITH_AES_128_CBC_SHA`, abbreviated as TLS-AES) and as recommended (`TLS_RSA_WITH_3DES_EDE_CBC_SHA`, abbreviated as TLS-3DES). Since the impact differences between these two cipher suites are mainly on the bulk data encryption phase, we test both cipher suites only in the proxy chain mode which is specifically meant to examine the cost of TLS bulk data encryption. For all other three proxy modes, we test TLS-AES only. Second, we enable SIP proxy authentication only in the outbound proxy and local proxy modes, which is a common setting. Third, we test the TLS session resumption and TLS mutual authentication separately to understand each of their impacts. We configure appropriate certificates on both servers and clients in the experiments which require them. Fourth, when both the left path and the right path can apply TLS session resumption or TLS mutual authentication, both paths are configured to have the same setting. These decisions reduce our test space for TCP and TLS to 16 configurations. Adding the two UDP with and without proxy authentication settings, we have a total of 18 test configurations.

## G. *Limitations and Scope*

Note that the performance measurements we obtained are, by practical necessity, limited to one implementation and platform and may not be directly applicable to other SIP server implementations and platforms. However, we believe that we chose a mature and commonly-deployed SIP server architecture on a fairly typical hardware platform, so these

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

TABLE I: Overall Test Matrix

| UDP NoAuth | TCP NoAuth | TLS AES NoAuth | TLS 3DES NoAuth | UDP Auth | TCP Auth | TLS AES Auth | TLS 3DES Auth |
|---|---|---|---|---|---|---|---|
| 2400 | 1139 | 695 | 534 | 462 | 361 | 276 | 244 |

TABLE II: Peak Throughput: Proxy Chain

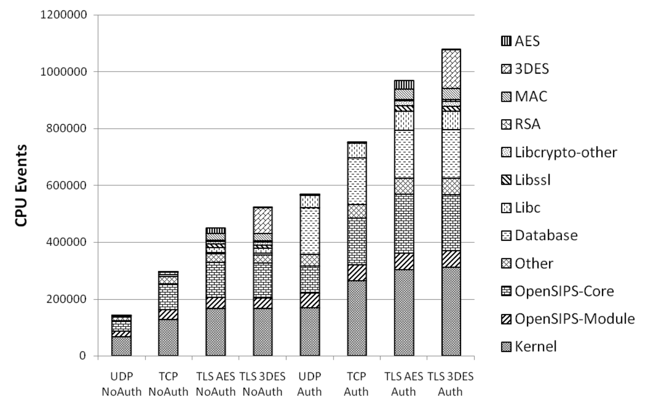

Fig. 5: CPU Profile Cycle Costs: Proxy Chain (50 cps)

measurements will provide a good indication of the kind of performance trade-offs that can be expected.

## IV. Results and Analysis

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

### A. *Proxy Chain*

Table II lists the peak throughput (T) in cps in the proxy chain mode. Each number is for a different configuration. The first four configurations have SIP proxy authentication disabled (NoAuth) and the next four have SIP proxy authentication enabled (Auth). The tests include UDP, TCP only, TLS with the TLS-AES cipher suite, and TLS with the TLS-3DES cipher suite. Recall that in this mode, no connection setup overheads are incurred. The average CPU utilization ranges from $95\%$ to $100\%$ in all the peak test cases except for the UDP and TCP without authentication cases, which is about $70\%$ and $85\%$, respectively. Although not all the tests could reach full CPU utilization because there is not always quite enough client machines to fully load the testbed, we take this factor into account in our cost model analysis in Section V by scaling on CPU utilization appropriately.

It can be seen that the peak throughput using TCP achieves about $47\%$ of the throughput using UDP, when SIP proxy authentication is not used. When the authentication is enabled, TCP provides $78\%$ of the corresponding UDP throughput. 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. Our approach is to obtain a CPU profile of each configuration run at the same load level of 50 cps 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 might not always be the case. To test the accuracy of this assumption, we compare the observed peak throughputs with the ones extrapolated based on the CPU cycle costs observed later in Section V-B.

Fig. 5 shows the number of non-idle CPU cycles consumed by the server in the proxy chain mode for each configuration during the test. To facilitate understanding, we also group the individual functions into several major categories. For example, the OpenSIPS server costs consist of basic core functions (OpenSIPS-Core) and those functions implemented as modules (OpenSIPS-Module) such as for stateful transactions, record route, and etc. For the detailed function vs. category mapping table please refer to Appendix C of our technical report [55]. From Fig. 5 we see that the total cost of the baseline UDP case without SIP authentication is about 144K CPU cycles. The most significant cost components are kernel (68K) which accounts for $47\%$, and the sum of OpenSIPS-Core and OpenSIPS-Module (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,

| UDP | TCP | TLS | TLS MutualAuth | TLS SessionResume |
|---|---|---|---|---|
| 462 | 268 | 151 | 122 | 181 |

TABLE III: Peak Throughput: Outbound Proxy

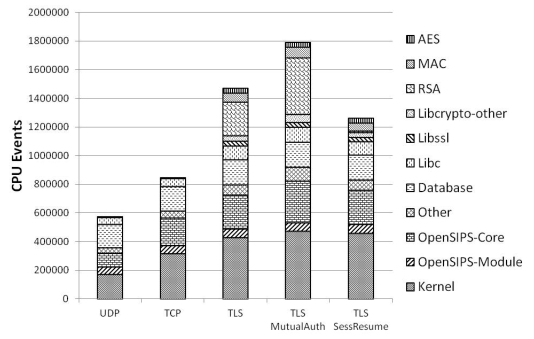

Fig. 6: CPU Profile Cycle Costs: Outbound Proxy (50 cps)

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

A major cost increase in scenarios with SIP proxy authentication compared to those scenarios without it is the database cost incurred when the SIP server accesses the records in the MySQL server during the authentication. This cost 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 $44\%$ overhead 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

Table III shows peak throughputs of the outbound proxy mode in different configurations, namely UDP, TCP, TLS, TLS with mutual authentication, and TLS where session resumption. Each configuration has SIP authentication enabled. Since the choice of AES or 3DES only affects the bulk data encryption overheads, which we have 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 in each case is around $90\%$.

Recall that in the TCP or TLS cases of this mode, each call results in a new connection being established with the server, as opposed to the proxy chain mode above. We see that the peak throughput in the TCP case is around $58\%$ of the baseline UDP case. The TLS case is approximately $56\%$ of the TCP case. Within the TLS cases, adding TLS mutual authentication reduces throughput about $20\%$, while enabling session resumption increases throughput about $20\%$.

Fig. 6 shows the CPU profiles for different outbound proxy configurations, again at the 50 cps 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 $75\%$ 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) 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), 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 cycles 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 [27]. In addition, the server performs another RSA public key decryption for the client's certificate verification message and also verifies the certificate using the MAC algorithm. Indeed, we see MAC costs increase by 10K cycles when mutual authentication is used. Kernel costs also increase by 45K cycles, presumably due to additional network packets transmitted and context switches between user and kernel space.

However, enabling TLS session resumption reduces the overhead by 200K cycles compared to the baseline TLS case. 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 resumption case, no key exchange and certificate verification is required. MAC costs remain, however, since new cryptographic keys are still computed for data encryption.

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

| Config | UDP | TCP | TLS | TLS MutualAuth | TLS SessionResume |
|---|---|---|---|---|---|
| Original | 2400 | 200 | 125 | 80 | 130 |
| W/TOfix | 2400 | 583 | 161 | 115 | 326 |

TABLE IV: Peak Throughput: Inbound Proxy

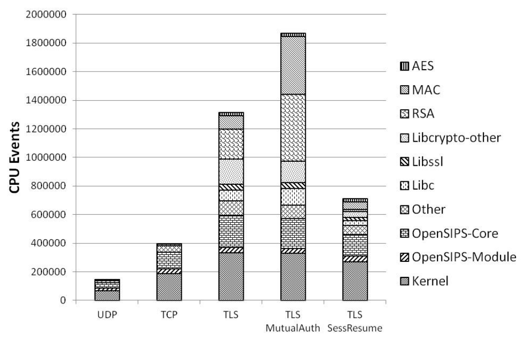

Fig. 7: CPU Profile Cycle Costs: Inbound Proxy (with Timeout Fix)

authenticate and authorize "users".

## C. Inbound Proxy

Table IV shows the peak throughput of the inbound proxy mode, where SIP proxy authentication is not enabled. The table shows two versions of OpenSIPS: the original version and one with a modification we developed, denoted as "W/TOfix". 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, 50% of the CPU cycles in the TCP case and 20% of the CPU cycles in the TLS case are spent in two functions called `tcp_main_loop` and `tcp_receive_loop`. More detailed profiling revealed that the overheads 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 existing server code calls a `timeout` function every time the `epoll` mechanism returns when events are detected. Since the connection expiration time is not linked to the corresponding hash key, during each call to the `timeout` function, the entire TCP connection hash table is traversed. Therefore, at high loads when the hash table has thousands of entries, the time spent in the timeout function becomes much larger than that of the case under lower load.

We applied a fix to the 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 already called the timeout function during the current time tick value, it will not enter the timeout function until the time tick value is advanced. This simple fix turned out to have a significant impact on performance involving TCP and TLS, as shown in Table IV.

As can be seen, the TCP case and the TLS with session resumption 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 cps 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%.

In the TLS and TLS with mutual authentication, the proportion of cryptographic overheads takes a greater part of the total cost. Therefore, although they also see performance increases from the timeout fix, the differences are less dramatic.

From Table IV, 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 resumption increases throughput by 100%.

Fig. 7 shows the CPU profiles for the 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-Module 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 resumption reduces costs by 46% compared to the base TLS case, with total costs falling from 1,315K to 710K cycles. Reduced RSA processing contributes 200K of those reductions; other libcrypto costs drop by 135K; MAC overheads are reduced by 40K; libssl costs shrink by 20K.

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

An interesting observation from Fig. 7 is the cost of MAC functions, which are substantially higher than in the previous proxy scenarios. This is because the proxy in the inbound mode acts as TLS client and needs to verify the certificates presented by the UAS, which was not present in the outbound mode. In addition, in the mutual TLS case, the inbound proxy needs to perform RSA encryption using its own private key and to sign the certificates using the MAC algorithm.

| Config | UDP | TCP | TLS | TLS MutualAuth | TLS SessionResume |
|---|---|---|---|---|---|
| Original | 462 | 136 | 65 | 60 | 100 |
| W/TOfix | 462 | 247 | 91 | 61 | 151 |

TABLE V: Peak Throughput: Local Proxy

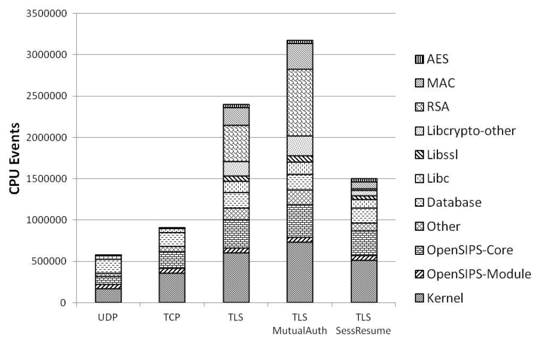

Fig. 8: CPU Profile Cycle Costs: Local Proxy (with Timeout Fix)

These overheads are exhibited in the profiles. Furthermore, in the TLS with session resumption 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.

Comparing Table IV and Table III, we also notice that in the UDP, TCP and TLS with session resumption cases, the throughputs in the outbound proxy mode are much higher than those in the inbound proxy mode. Fig. 7 and Fig. 7 reveal that this is mainly due to the additional database lookup cost during user authentication in the outbound proxy mode. In the other two cases, TLS and TLS with mutual authentication, the TLS associated costs take a larger part and flatten out the impact of the additional database cost, therefore the throughput differences of these two modes are not as noticeable. It is worth noting that using a more efficient database module such as an in-memory database is expected to significantly reduce the database cost.

## D. Local Proxy

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

The average CPU utilizations in the four configurations with the timeout fix are between $80\%$ to $90\%$. We see that the peak throughput with TCP is around $53\%$ of the UDP case, while the peak throughput with TLS is approximately $37\%$ of the TCP case. Within the TLS cases, adding TLS mutual authentication reduces throughput by $33\%$, while enabling session resumption increases throughput by $66\%$.

Fig. 8 shows the CPU profile results for the local proxy mode with the timeout fix. In general, the use of TCP incurs $58\%$ additional overhead compared to the baseline UDP case. 186K of this is contributed by Kernel, 108K by OpenSIPS-Core and OpenSIPS-Module, 10K by libc and 30K by other functions. Using TLS introduces over $166\%$ of additional overhead compared to the TCP case. Total cycles increase by 1,500K from 900K to 2,400K. RSA contributes 434K to that increase, followed by kernel overheads 240K, MAC processing 219K, other libcrypto functions 174K, OpenSIPS-Core 140K, libssl 67K, and AES 36K.

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

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

The MAC cost is significantly reduced in the TLS with session resumption 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 measurement-driven cost model derived from results of all the 18 test scenarios in Section IV. The model shows the overall cost relationship among the different configuration parameters. While clearly performance will vary across systems, our model helps network administrators in provisioning their systems by providing an intuitive way to estimate where the overheads in deploying SIP-over-TLS are and gives guidance on relative performance across a single system with and without TLS. 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 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 mode configuration incurs at a load of 50 cps. This 50 cps is a load level that can be sustained by all the test configurations. The decomposition of costs is enabled by our characterization of the four SIP server connection modes (chain, outbound, inbound, local proxy) as in Section III-F because each of the first three connection modes allows us to examine a different aspect of the system in terms of TLS cost. For the proxy chain mode, since there is no additional connection establishment cost once the first connection has been ready, it allows us to solely evaluate the cost impact incurred

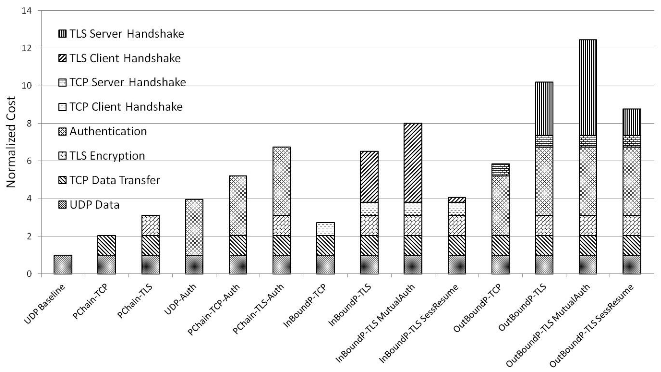

Fig. 9: Functional Components Cost Model

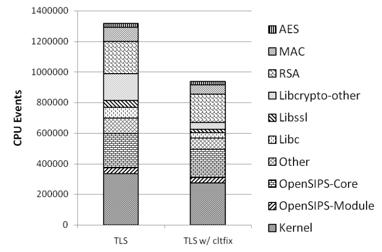

Fig. 10: Impact of TLS Client Side Fix on Inbound Proxy

in TLS bulk data encryption. The outbound and inbound proxy modes 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 local proxy mode gives us an overall view combining all the aspects involved in the first three modes.

In deriving the model, we start from the CPU event cost of the baseline configuration in proxy chain mode with UDP. This cost is shown in the first bar in Fig. 5 we normalize it as one cost unit and use it as the base for computing other costs. For example, the second bar in Fig. 5 indicates a cost of 2.1 units for the proxy chain mode with TCP. Then, the difference of these two costs, 1.1, gives us the incremental overhead of using TCP for data transfer. 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 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 mode configurations. These costs are plotted in Fig. 9. Below we explain in more detail each of the functional components in Fig. 9 and compare them in difference scenarios.

The UDP Data 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. At a cost of 1.1, it is 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.

The Authentication cost represents the cost of the SIP proxy 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 60% and 70% 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. Surprisingly, we originally observed that the TLS client side cost is 90% higher than its server side cost in both the TLS and TLS mutual authentication scenarios, which is contrary to the common wisdom [45]. We looked into the code and found an OpenSSL-related redundant code path traversal in the OpenSIPS server. After applying another fix, Fig. 10 shows the new CPU cycle cost as opposed to the original cost in the inbound proxy TLS mode. As can be seen, the CPU costs shrink in virtually all categories except AES encryption cost, which should not be affected anyway. To further verify the TLS library cost associated with establishing TLS connections (as TLS client) and accepting TLS connections (as TLS server), we compare the corresponding cost incurred in the SIP proxy with that incurred in a simple HTTPS client server

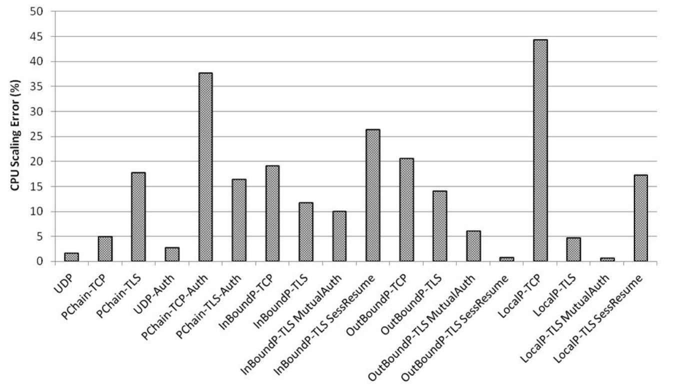

Fig. 11: CPU Scaling Error within Each Proxy Configuration

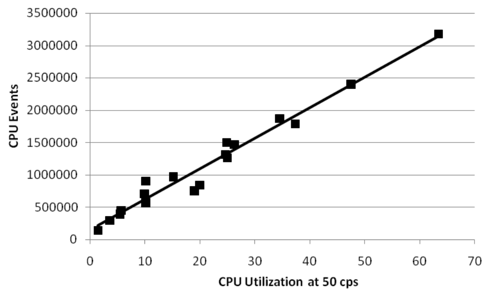

Fig. 12: CPU Events vs. CPU Utilization Across Proxy Configurations

application [44], assuming a similar number of connections are being set up. Results indicate a 10% to 15% reasonably close crypto costs match between the complex SIP proxy server and the simple HTTPS server. Therefore, the numbers we list in Fig. 9 are values with our TLS client side connection fix applied. With the normal TLS handshake, the cost at the client side and server side are 2.7 and 2.9 respectively. When TLS mutual authentication is enabled, the cost at the client and server side nearly doubles at 4.2 and 5.1 respectively. With TLS session resumption, the TLS client side cost reduces by 90% to 0.26 and the TLS server side cost shrinks by 50% to 1.4. These numbers represent a TLS client side cost reduction of 50%, 55% and 73% in TLS, TLS mutual authentication and TLS session resumption cases compared with the original unpatched server. In other words, it means considerable throughput improvements in those operation modes.

## B. Validating the Component Cost Model

We took two steps to validate that our component cost model derived at a particular load point of 50 cps can be scaled to higher load values as well.

The first step is to validate that, within a particular proxy mode configuration, 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 Fig. 11. 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 validate the model is to check that the relationship between CPU events and CPU utilization is also linear, because we want to use the cost model to predict peak throughput relationships across different scenarios. From our experience we know that the number of CPU events is a more stable estimate than CPU utilization, which has higher variability, particularly at low loads. If the events and CPU utilization across different scenarios exhibit a linear relationship, and since we know from above that the throughput scales linearly with CPU utilization, we can similarly scale the event cost within each scenario. This lets us obtain a predicted peak throughput relationship across different scenarios by taking the inverse of the cost for each scenario at the 50 cps load level. Fig. 12 shows the number of CPU events vs. CPU utilization across all 18 of our peak throughput measurements. The Y-axis presents the CPU events 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 observed in step one of the validation.

## C. Using the Component Cost Model

Our 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. After comparing the model-derived costs and the actual measured costs in the local proxy mode, we found that the two sets of costs differ within 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.

The second and more common use of the functional cost model is to approximate the peak throughput of different scenarios. Since the CPU cost and utilization scale relatively linearly according to the load as we have seen in our model validation, the peak throughput should be inversely proportional to the cost. Therefore, if we know the peak throughput of the baseline UDP proxy chain mode configuration, we will be able to project the peak throughput of other scenarios with different configuration combinations.

For example, from Fig. 9, we know that depending on whether SIP proxy authentication is enabled, the use of TCP reduces throughput by 51% or 24% over UDP in the proxy chain mode. When TCP connection handshake and maintenance costs are incurred as in other scenarios, the throughputs drop by 64% in the inbound mode and 32% to 40% in the outbound and local proxy modes compared to UDP. When comparing normal TLS versus plain TCP, if only bulk data encryption is used as in the proxy chain mode, the model suggests that TLS with AES reduces throughput by 34% to 22% depending on whether SIP proxy authentication is enabled. When both bulk data encryption and TLS handshake costs are incurred as in the other proxy modes, the use of TLS reduces throughput by 58% to 52% in the inbound 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 22% depending on the proxy mode. When TLS session resumption is enabled, the throughput is increased by 16% in the outbound proxy mode and by 40% to 60% in the inbound and local proxy modes.

## VI. Related Work

SSL/TLS performance has been studied by a number of researchers. However, almost all these studies are based on SSL/TLS Web servers. Apostolopoulos et al. [10] 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. [33] 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. [63] provided an oprofile-based anatomy of SSL processing for an SSL Web server. They found that about 70% of the total processing time of an HTTP-over-SSL transaction is spent in SSL processing. Coarfa et al. [15] measured the difference of TLS server throughput by selectively replacing TLS operations with no-ops, instead of using a CPU profiler. Their results show that RSA computations are the single most expensive operation in TLS, which accounts for $13-58\%$ of the total time spent under different available server CPU cycles and workload conditions.

Zeng and Cherkaoui [62] studied the performance of TLS on a Common Open Policy Service (COPS) over TLS environment. Their results showed that the cost of establishing a COPS-over-TLS session took about a thousand times what is needed for a plain COPS connection without TLS.

Many researchers have studied SIP server performance, albeit without TLS. Schulzrinne et al. presented SIPstone [54], a suite of SIP benchmarks for measuring SIP server performance on common tasks. Cortes et al. [17] measured the performance of four different stateful SIP proxy server implementations over UDP and reported throughput results from $90-700$ cps. Nahum et al. [21], [36] showed experimental performance results of the OpenSER SIP server under different scenarios including stateful and stateless proxying, TCP and UDP transport, with and without SIP proxy authentication. Their results indicate that any of these configurations can affect performance by a factor of $2-4$. Their evaluated SIP-over-TCP scenario corresponds to the TCP single connection in the proxy chain mode of this paper. Ono and Schulzrinne [40] studied the performance of the SIPd [52] SIP server over the UDP and TCP transports. Their TCP tests include the multiple connection mode between the SIP proxy and the UA similar to the local proxy mode of this paper. Ram et al. [42] provided more understanding of the impact of TCP on SIP server performance using OpenSER. They showed that a substantial component of the performance loss from using TCP is due to the process architecture in OpenSER and provided improvements. Wright et al. [61] studied the performance of SIP servers on multi-core systems. They proposed and evaluated several optimizations to improve scalability up to eight cores. In addition, a number of researcher have looked into the SIP server performance over UDP or TCP under overload conditions, such as Shen et al. [56], [57], Hilt et al. [30], [31], Noel et al. [39] and Abdelal et al. [8].

Cha et al. [14] described a study of SIP with TLS, DTLS and IPsec over TCP, UDP and SCTP. The work is based on ns-2 [3] simulation and the scope of the evaluation is on call setup delay in a two-hop SIP proxy scenario with background traffic. Thus the focus is on delay as a function of packet exchanges rather than server CPU overheads. Kim et al. [35] measured the call setup delay (along with voice quality metrics such as mean option score) on a SIP-based VoIP system which contains both TLS and S-MIME. The paper indicates that the testbed was built by the authors although it is not clear what the exact software and hardware used are, or what the call request rate during the measurement is. Gurbani et al. [29] proposed a different SIP-over-TLS approach with cryptographically transparent SIP proxy servers. They use a newly defined SIP `CONNECT` method to establish a TCP overlay between the SIP UAs directly and allow them to then establish a TLS session directly. Thus, intermediate proxy servers merely forward encrypted messages which protects the signaling privacy. Their empirical results also showed noticeable latency and significant CPU cost compared to existing SIP-over-TLS usage, due to reduced overhead in session renegotiation and cryptographic operations. On another study, Subramanian and Dutta [58] conducted an experimental study of SIP proxy server performance with TLS based on CISCO SIP proxy server and CISCO Camelot call generators. Their testbed comprises two servers acting as outbound and inbound proxy servers, respectively. They reported call setup delay, overall CPU utilization, average queue size and memory cost of the testbed at a 1,000 cps load. In contrast, our study considered a more comprehensive set of SIP proxy configurations, different possible TLS operation modes, and detailed server CPU profiling. Yet another relevant work in SIP-over-TLS performance is from Salsano et al. [50] who measured the throughput and processing cost of SIP proxy server over UDP, TCP and also TLS. Their test cases for stateful SIP proxy servers represent four of the 18 scenarios that we look at, essentially the UDP NoAuth, UDP Auth, TCP Auth, and TLS Auth configurations, all in the proxy chain mode. The total cost ratios of these four scenarios in their work are 1:1.44:1.52:1.54, while the corresponding ratios from our results are 1:4:5.2:6.7. These numbers are not directly comparable because of the different software and hardware platforms used in the two sets of experiments. Salsano et al. used their own open-source SIP server implemented in Java with 300 MHz Pentium machines running either Linux or Windows 98/2000. We use contemporary hardware and standard open-source software implemented in C. As a result, the peak performance of the two testbeds are also dramatically different. For example, in the basic UDP NoAuth scenario, the peak throughput on their testbed is 21 cps, compared to 2,400 cps on ours, a factor of 100 difference in performance.

One approach to reducing security overheads is to use a hardware crypto accelerator, e.g., Sun's Crypto 6000 card [16]. While this can improve performance (e.g., the card claims 13,000 1024-bit RSA operations per second), the cards tend to be expensive (e.g., the list price for the board was $9,950 at the time of this writing). More importantly, in many cases, much of the overhead we observed was in the OpenSSL software libraries themselves (e.g., libssl, libssl-other), rather than the crypto algorithms (libcrypto). Crypto acceleration hardware will not help with these overheads.

## VII. Conclusions

Insecure UDP-based signaling is one major reason that exposes SIP-based services to many common security threats. We have evaluated and analyzed the impact of using TLS as a transport on SIP server performance versus the standard approach of SIP-over-UDP. Using an experimental testbed with the OpenSIPS server, OpenSSL, Linux, and an Intel-based server, we show that TLS can reduce SIP server performance 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 resumption is effective).

In the best case, the baseline UDP performance is about three times that with TLS (proxy chain); in the worst case, UDP is 17 times the performance than with TLS (local proxy with TLS 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 resumption). 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 issues with OpenSIPS and OpenSSL, despite the fact that they have mature code bases and large numbers of users. These issues were usually overlooked because they only manifest themselves in high-load, multiple-connection scenarios. The fixes to these problems range from straightforward adjustment of default parameter values, to more complicated code path optimization and rather subtle library bug patches. When these fixes are applied, performance improved in some cases from a few times up to an order of magnitude.

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

[2] MySQL. http://www.mysql.org.

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

[4] OpenSSL. http://www.openssl.org.

[5] OProfile. http://oprofile.sourceforge.net.

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

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

[8] A. Abdelal and W. Matragi. Signal-based overload control for SIP servers. In *Proc. 7th Annu. IEEE Consumer Commun. and Networking Conf.*, Las Vegas, Nevada, Jan. 2010.

[9] A. Acharya, X. Wang, and C. Wright. A programmable message classification engine for Session Initiation Protocol (SIP). In *Proc. 3rd ACM/IEEE Symp. Architecture for networking and commun. syst.*, pages 185–194, Orlando, FL, Dec. 2007.

[10] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Proc. 18th Annu. Joint Conf. IEEE Comput. and Commun. Soc.*, New York, NY, Mar. 1999.

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

[12] D. Butcher, X. Li, and J. Guo. Security challenge and defense in VoIP infrastructures. *IEEE Trans. Syst., Man, and Cybern., Part C: Applicat. and Reviews*, 37(6):1152–1162, Nov. 2007.

[13] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitima, and D. Gurle. Session Initiation Protocol (SIP) extension for instant messaging. RFC 3428, Dec. 2002.

[14] E. Cha, H. Choi, and S. Cho. Evaluation of security protocols for the Session Initiation Protocol. In *Proc. 16th Int. Conf. on Comput. Commun. and Networks*, Honolulu, HI, Aug. 2007.

[15] C. Coarfa, P. Druschel, and D. Wallach. Performance analysis of TLS Web servers. In *Proc. Internet Soc. Symp. on Network and Distributed Syst. Security*, San Diego, CA, Feb. 2002.

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

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

[18] I. Dacosta, V. Balasubramaniyan, M. Ahamad, and P. Traynor. Improving authentication performance of distributed SIP proxies. In *Proc. 3rd Int. Conf. Principles, Syst. and Applicat. of IP Telecomm*, pages 1–11, Atlanta, GA, Jul. 2009.

[19] I. Dacosta and P. Traynor. Proxychain: Developing a robust and efficient authentication infrastructure for carrier-scale VoIP networks. In *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, Jun. 2010.

[20] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, Aug. 2008.

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

[22] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, Sep. 2001.

[23] J. Fabini, N. Jordan, P. Reichl, A. Poropatich, and R. Huber. "IMS in a bottle": initial experiences from an OpenSER-based prototype implementation of the 3GPP IP multimedia subsystem. In *Proc. Int. Conf. Mobile Bus.*, page 13, Copenhagen, Denmark, Jun. 2006.

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

[25] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet draft, Netscape Communications, Nov. 1996. http://wp.netscape.com/eng/ssl3/ssl-toc.html.

[26] R. Gayraud and O. Jacques. SIPp. http://sipp.sourceforge.net.
[27] V. Gurbani, S. Lawrence, and A. Jeffrey. Domain certificates in Session Initiation Protocol (SIP). RFC 5922, Jun. 2010.
[28] V. Gurbani, R. Mahy, and B. Tate. Connection reuse in the Session Initiation Protocol (SIP). RFC 5923, Jun. 2010.
[29] V. Gurbani, D. Willis, and F. Audet. Cryptographically transparent Session Initiation Protocol (SIP) proxies. In *Proc. IEEE Int. Conf. on Commun.*, pages 1185 –1190, Jun. 2007.
[30] V. Hilt, E. Noel, C. Shen, and A. Abdelal. Design considerations for Session Initiation Protocol (SIP) overload control. RFC 6537, Aug. 2011.
[31] V. Hilt and I. Widjaja. Controlling overload in networks of SIP servers. In *Proc. IEEE Int. Conf. on Network Protocols (ICNP)*, Orlando, Florida, Oct. 2008.
[32] IPTel.org. SIP Express Router (SER). http://www.iptel.org/ser.
[33] K. Kent, R. Iyer, and P. Mohapatra. Architectural impact of secure socket layer on Internet servers. In *Proc. Int. Conf. Comput. Design*, pages 7–14, Austin, TX, Oct. 2000.
[34] A. Keromytis. Voice over IP: Risks, threats and vulnerabilities. In *Proc. Cyber Infrastructure Protection Conf.*, New York, NY, Jun. 2009.
[35] J. Kim, S. Yoon, H. Jeong, and Y. Won. Implementation and evaluation of SIP-based secure VoIP communication system. In *Proc. IEEE/IFIP Int. Conf. Embedded and Ubiquitous Computing*, Shanghai, China, Dec. 2008.
[36] E. Nahum, J. Tracey, and C. Wright. Evaluating SIP proxy server performance. In *Proc. 17th Int. Workshop Networking and Operating Syst. Support for Digital Audio and Video*, Urbana-Champaign, IL, Jun. 2007.
[37] NIST. Data Encryption Standard (DES), Dec. 1993. http://www.itl.nist.gov/fipspubs/fip46-2.htm.
[38] NIST. Advanced Encryption Standard (AES), Nov. 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
[39] E. Noel and C. Johnson. Novel overload controls for SIP networks. In *Proc. 21st Int. Teletraffic Congr.*, Paris, France, Sep. 2009.
[40] K. Ono and H. Schulzrinne. One server per city: using TCP for very large SIP servers. In *Proc. 2nd Int. Conf. Principles, Syst. and Applicat. of IP Telecomm*, pages 133–148, Heidelberg, Germany, Oct. 2008.
[41] OpenSIPS. The open SIP server. http://www.opensips.org.
[42] K. Kumar Ram, I. Fedeli, A. Cox, and S. Rixner. Explaining the impact of network transport protocols on SIP proxy performance. In *Proc. IEEE Int. Symp. Performance Anal. of Syst. and Software*, pages 75–84, Austin, TX, Apr. 2008.
[43] Light Reading. VoIP security: vendors prepare for the inevitable. *VoIP Services Insider*, 5(1), Jan. 2009.
[44] E. Rescorla. openssl-examples. http://www.rtfm.com/openssl-examples.
[45] E. Rescorla. *SSL and TLS: designing and Building Secure Systems*. Addison Wesley, 2000.
[46] E. Rescorla and N. Modadugu. Datagram transport layer security. RFC 4347, Apr. 2006.
[47] R. Rivest. The MD5 message digest algorithm. RFC 1321, Apr. 1992.
[48] R. Rivest, A. Shamir, and L. Adleman. Cryptographic commun. syst. and method. Technical Report TR-212, MIT Lab for Computer Science, Jan. 1979.
[49] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Jun. 2002.
[50] S. Salsano, L. Veltri, and D. Papalilo. SIP security issues: the SIP authentication procedure and its processing load. *IEEE Network*, 16(6):38–44, Nov. 2002.
[51] B. Schneier. *Applied Cryptography (2nd Edition)*. John Wiley and Sons, New York, NY, 1996.
[52] H. Schulzrinne. SIPd. http://www.cs.columbia.edu/IRT/cinema.
[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. RFC 3550, Jul. 2003.
[54] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone - benchmarking SIP server performance. http://www.sipstone.com.
[55] C. Shen, E. Nahum, H. Schulzrinne, and C. Wright. The impact of TLS on SIP server performance. Technical Report cucs-022-09, Columbia Univ. Computer Science Dept., 2009.
[56] C. Shen and H. Schulzrinne. On TCP-based SIP server overload control. In *Proc. Principles, Syst. and Applicat. of IP Telecomm*, pages 71–83, Munich, Germany, Aug. 2010.
[57] C. Shen, H. Schulzrinne, and E. Nahum. Session Initiation Protocol (SIP) server overload control: design and evaluation. In *Proc. Principles, Syst. and Applicat. of IP Telecomm (IPTComm). Services and Security for Next Generation Networks*, volume 5310/2008, pages 149–173, Oct. 2008.
[58] S. V. Subramanian and R. Dutta. Comparative study of secure vs. non-secure transport protocols on the SIP proxy server performance: an experimental approach. In *Proc. Int. Conf. Advances in Recent Technologies in Commun. and Computing*, pages 301–305, Kottayam, India, Oct. 2010.
[59] V. Tzvetkov and H. Zuleger. Service provider implementation of SIP regarding security. In *Proc. 21st Int. Conf. Advanced Inform. Networking and Applicat. Workshops*, volume 1, pages 30–35, Niagara Falls, Canada, May 2007.
[60] X. Wang, R. Zhang, X. Yang, X. Jiang, and D. Wijesekera. Voice pharming attack and the trust of VoIP. In *Proc. 4th int. conf. Security and privacy in commun. netowrks*, pages 1–11, Istanbul, Turkey, Sep. 2008.
[61] C. Wright, E. Nahum, D. Wood, J. Tracey, and E. Hu. SIP server performance on multicore systems. *IBM J. Research and Develop.*, 54(1), Feb. 2010.
[62] Y. Zeng and O. Cherkaoui. Performance study of COPS over TLS and IPsec secure session. In *Proc. 13th IFIP/IEEE Int. Workshop Distributed Syst.: Operations and Manage.*, pages 133–144, Montreal, Canada, Oct. 2002.
[63] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of SSL processing. In *Proc. Int. Symp. Performance Anal. of Systems and Software*, pages 197–206, Austin, TX, Mar. 2005.

**Charles Shen** holds Ph.D. and M.S. degrees from Columbia University in the City of New York, as well as M.Eng. and B.S. degrees from National University of Singapore and Zhejiang University of China. He is a Senior Member of Technical Staff at AT&T Security Research Center in New York City. Prior to AT&T, he conducted research at Columbia University Computer Science Department, IBM Watson Research Center, Telcordia Technologies, Samsung Advanced Institute of Technology, and Institute for InfoComm Research of Singapore.

Dr. Shen's research interests include next generation IP telecommunications, mobile applications and services, and the Internet of Things. Dr. Shen is also an active contributor to international standardization bodies such as the Internet Engineering Task Force (IETF).

**Erich Nahum** is a research staff member at the IBM T.J. Watson Research Center. He received his Ph.D. in Computer Science from the University of Massachusetts, Amherst in 1996. He is interested in all aspects of performance in experimental networked systems.

**Henning Schulzrinne** , Levi Professor of Computer Science at Columbia University, received his Ph.D. from the University of Massachusetts in Amherst, Massachusetts. He was an MTS at AT&T Bell Laboratories and an associate department head at GMD-Fokus (Berlin), before joining the Computer Science and EE departments at Columbia University. He served as chair of the Department of Computer Science from 2004 to 2009 and as Engineering Fellow at the US Federal Communications Commission (FCC) in 2010 and 2011.

Protocols co-developed by him, such as RTP, RTSP and SIP, are now Internet standards, used by almost all Internet telephony and multimedia applications. His research interests include Internet multimedia systems, ubiquitous computing, and mobile systems. He is a Fellow of the IEEE.

**Charles P. Wright** is a research staff member at the IBM T.J. Watson Research Center.