# UVOIP: CROSS-LAYER OPTIMIZATION OF BUFFER OPERATIONS FOR PROVIDING SECURE VOIP SERVICES ON CONSTRAINED EMBEDDED DEVICES

Dinil.D[1], Aravind.P.A[1], Thothadri Rajesh[1], Aravind.P[1], Anand.R[1], Jayaraj Poroor[2]

[1]Electronics and Communication department
Amrita School of Engineering
Amritapuri,Kollam690525,
Kerala.
{dinildivakar,aravind.p.a,thothadri.rajesh,aravind.nature,anandamrit8}@gmail.com
[2]Center for Cyber security
Amrita Vishwa Vidyapeetham
Amritapuri, Kollam690525,
Kerala.
jayaraj@arl.amrita.edu

***ABSTRACT***

*In this paper, we present an optimized implementation of secure VoIP protocol stack so that the stack would fit into the memory and computation budget of constrained embedded systems. The novel approach that we take to achieve this is to perform cross-layer optimization of buffers and buffer operations. Buffers and buffer operations are involved in playback, capture, codec transformations, and network I/O. Following this approach, we have implemented VoIP application functions, RTP, and Secure RTP protocols in a tightly integrated and highly optimized manner, on the top of the embedded TCP/IP stack, uIP. We call the protocol stack thus constructed, the uVoIP stack. We have tested the uVoIP stack in GNU/Linux Operating System using tunnel device for sending and receiving raw packets.*

***KEYWORDS***

*VoIP security, protocol stack, RTP, uIP, SRTP, UDP, IP, buffers, cross-layer optimization, integration, constrained embedded devices.*

## 1. INTRODUCTION

Voice over Internet Protocol (VoIP) – the transmission of voice over packet-switched IP networks is increasingly replacing the traditional telephone systems. VoIP offers cheaper communication and enhanced services to users, but suffers from all the vulnerabilities of IP-based services. For media transport, the standard protocol used is Real-time Transport Protocol (RTP) [1] which is susceptible to several attacks. Secure Real-time Transport Protocol (SRTP) [2] may be used to provide security against these attacks. However, implementation of an SRTP-based VoIP stack in a naive manner may be unsuitable for use with constrained embedded devices (limited RAM space and processing speed) owing to relatively high resource requirements. To overcome this, we propose a cross-layer approach to implement SRTP-based secure VoIP stack so that buffer requirements and buffer operations are minimized. Our challenge lies in optimizing the stack operations that involve maintaining buffers at different layers for voice transport functions and cryptographic functions, and minimizing time-consuming CPU-bound buffer operations that will drain the precious battery-resources. Our

optimized stack which we call uVoIP stack, is built on the top of uIP which is an open source IP stack[3] suitable for use in constrained embedded environment. The uVoIP stack optimizes buffer usage for: audio playback, audio capture, and network I/O. To construct the uVoIP stack we performed cross-layer optimization of buffer usage at various layers in the stack, viz. uVoIP, SRTP, RTP, and Application functions (playback/capture).

The architecture diagram is shown in Figure 1. The same buffer used in the digitization phase is handed down the different layers without any copy-operations and is finally used to send the data out via the network interface (in our case, we used tunnel device to output raw IP packets). Similarly, the buffer used to store an incoming packet is passed up the layers without any copy-operations, and is finally used to play the voice via speaker output. Once buffers are freed after network I/O or playback, they are maintained in a 'free pool' and reused when new buffers are required by subsequent operations.

The structure of our paper is as follows. A brief introduction is given to RTP and SRTP in sections 2 and 3 respectively. The sections 4 and 5 deals with uIP architecture and tunnel interface respectively. In section 6 we describe about our novel attempt of the buffer minimization in uVoIP. Section 7 describes code design. In section 8 we present performance results from our experiments. Finally we summarize our conclusions and future work in section 9.

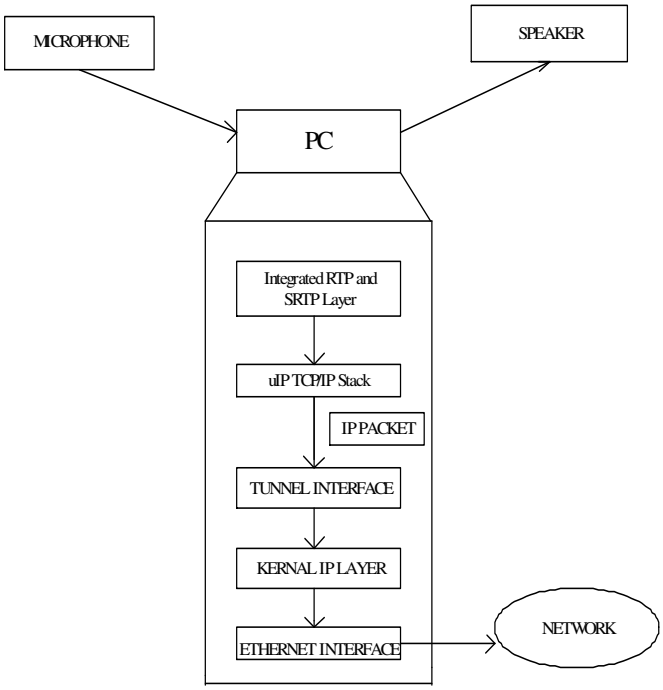


Figure 1. System Architecture Diagram

## 2. REAL-TIME TRANSPORT PROTOCOL (RTP)

For generating voice packets, we capture audio from microphone using /dev/dsp Linux device-special file with a 20 ms capture buffer. The sampling rate and sample size are fixed to 8000Hz and 8 bits/sample respectively for PCM encoding. The PCM encoding generates 160 bytes in 20 ms which has to be transmitted as RTP payload. The voice samples are inserted into data packets to be carried on the Internet along with the header fields and thus become RTP packets which hold data needed correctly to reassemble the packets on the other end .The control flow chart of RTP packet processing in sender side is shown in Figure 2.

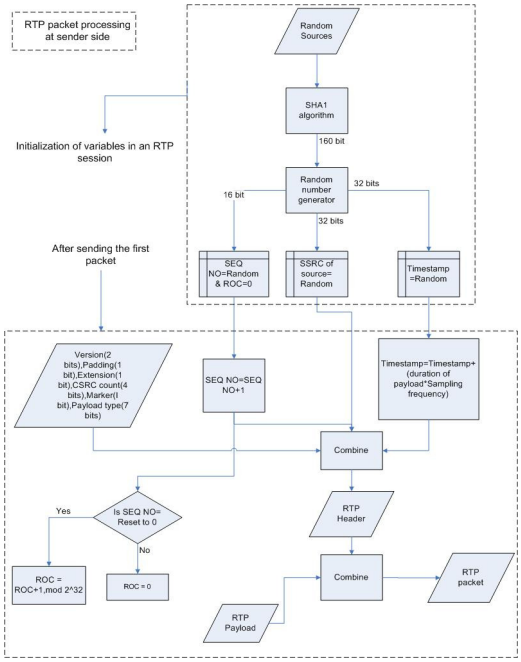


Figure 2. RTP packet processing in sender side

In the reception side, the RTP packets are appropriately processed after checking the validity of header field and the source which is explained with the help of control chart as shown in the Figure 3.

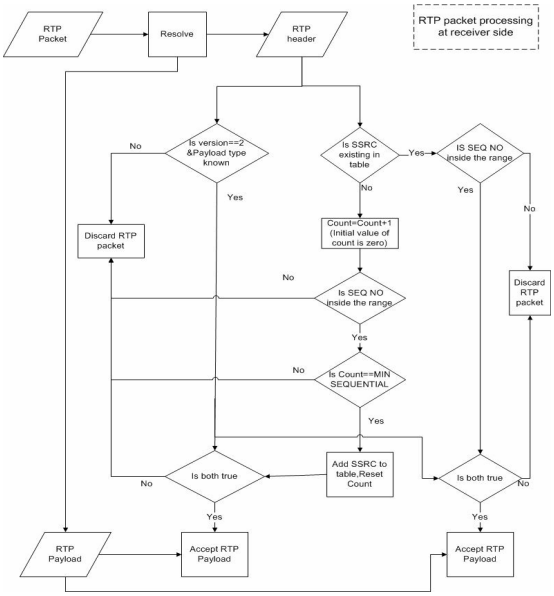


Figure 3. RTP packet processing in receiver side

## 3. SECURE REAL-TIME TRANSPORT PROTOCOL (SRTP)

Secure Real-time Transport Protocol (SRTP), is a profile of the Real-time Transport Protocol (RTP), which provides confidentiality, message authentication, and replay protection to the RTP traffic. SRTP has a set of default cryptographic transforms for providing encryption and a

function based on keyed-hash for message authentication. It also uses an implicit index for sequencing/synchronization based on the RTP sequence number.

SRTP encrypts only the payload (i.e., the audio or video) which provides confidentiality. The integrity of the entire original RTP packet is provided by the authentication algorithm. The only SRTP additions to the original RTP packet are the optional Master Key Identifier (MKI) which identifies the master key that was used to derive session keys currently in use for encryption/authentication and the recommended authentication tag (10 bytes) which provides protection against replay and ensures that neither the original RTP packets are not modified nor additional packets are inserted. The entire RTP packet is protected by performing encryption followed by authentication.

The data flow diagram in Figure 4 explains the SRTP packet processing at the sender side.

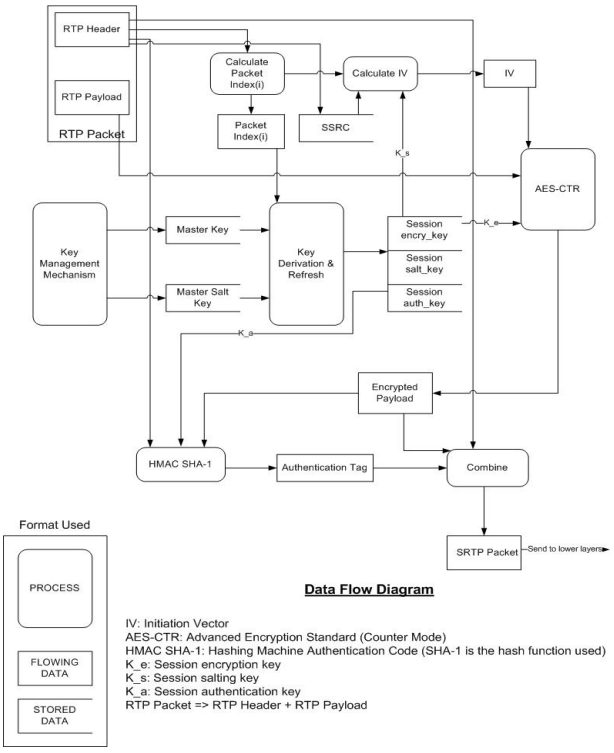


Figure 4. SRTP packet processing at the sender side

At the receiver side, the encrypted payload of received SRTP packet is used to calculate the authentication tag. If it matches with the authentication tag in the header, then the packet is the authentication check succeeds and packet is decrypted. The unsuccessfully authenticated packets are dropped. The Advanced Encryption Standard Counter Mode (AES-CTR) is the default encryption method used in SRTP [4]. A major reason that AES-CTR was chosen because there is no payload expansion produced (the encrypted payload is of the same length as the original payload) and also it can be used in parallel packet processing. SRTP uses a single master key to create all the required authentication and encryption keys. For doing this, it relies on a key derivation algorithm based on AES-CTR. Authentication is done using HMAC-SHA1 algorithm [5].

The session keys, obtained by key derivation mechanism consists of mainly three keys, namely

1. Session encryption key (k_e): It is 128 bit and the label used for its derivation is 0x00.
2. Session authentication key (k_a). It is 160 bit and the label used is 0x01.

3. Session salt key (k_s): It is 112 bit and the label used is 0x02.

The master key (128 bit) and master salt (112 bit) must be random, but the master salt may be public. Key derivation rate is 0 bit and we can set its value. According to that value, the refreshment of session keys occurs. The inputs to AES is k_e and the Initiation vector (IV) which is calculated using the formula given below

IV = (k_s * 2^16) XOR (SSRC * 2^64) XOR (i * 2^16)

## 4. UIP TCP/IP STACK

The uIP open source TCP/IP stack provides minimal set of features needed for a functional well-behaved TCP/IP stack. uIP is written in C with the code size in the order of a few kilobytes and the RAM usage can be made as low as a few hundred bytes which makes it suitable even for small 8-bit micro- controllers. It can only handle a single network interface and contains the IP, ICMP, UDP and TCP protocols.

The uIP stack uses single global packet buffer which is used for incoming packets and also for the TCP/IP headers of outgoing data. The global packet buffer is large enough to contain one packet of maximum size. The behavior of uIP stack is shown in Figure 5.

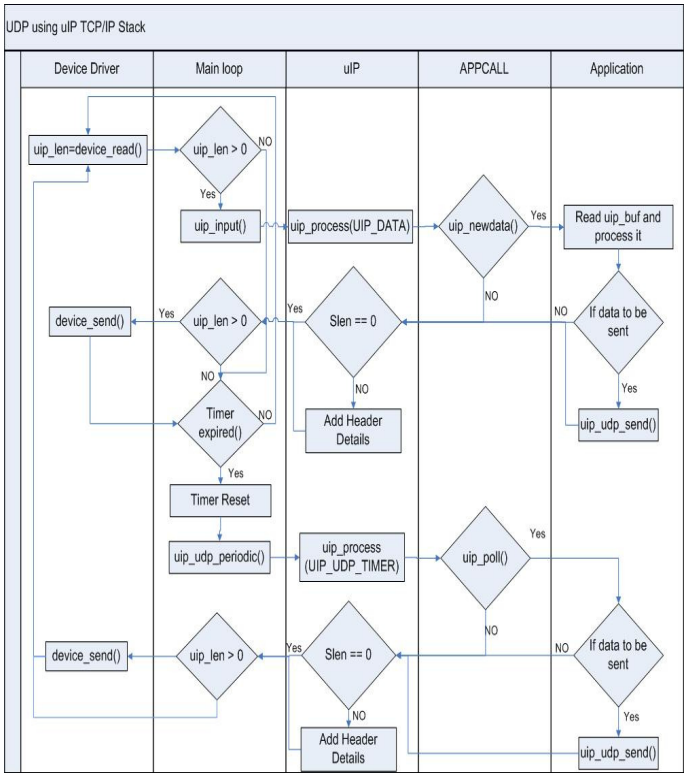


Figure 5. Description of uIP stack behavior

## 5. TUNNEL INTERFACE

The tunnel interface (TUN) allows reception and transmission of raw IP packets by user space programs. To quote from [6]: "It can be viewed as a simple Point-to-Point or Ethernet device, which instead of receiving packets from a physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program".

When a program opens /dev/net/tun, the driver creates and registers corresponding net device tunX. After a program closed above devices, the driver will automatically delete tunXX device and all routes corresponding to it. The tunnel interface is described in Figure 6.

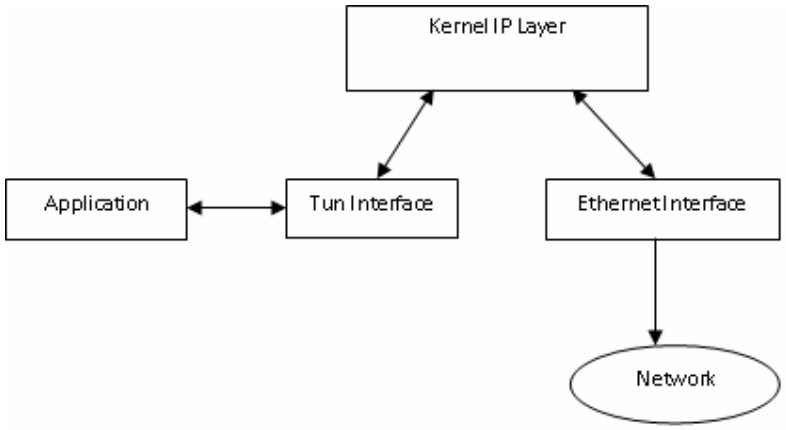


Figure 6. Tunnel Interface

When the application sends IP packets to the tunnel interface it will appear to the kernel as if it is coming from a physical interface (even though it is only a virtual interface). Therefore kernel IP layer routes the packet based on the kernel routing table. It may be noted that for route packets, the IP forwarding option should be enabled on the machine. Similarly the packets that kernel forwards to the tunnel interfaces could be read by our application and processed.

## 6. BUFFERS IN VOIP

In uVoIP stack, buffers are required for playback, capture and network I/O. These buffers are shared across different layers and allocated dynamically for minimizing number of buffers and buffer operations. The buffer usage in uVoIP is shown in Figure 7.

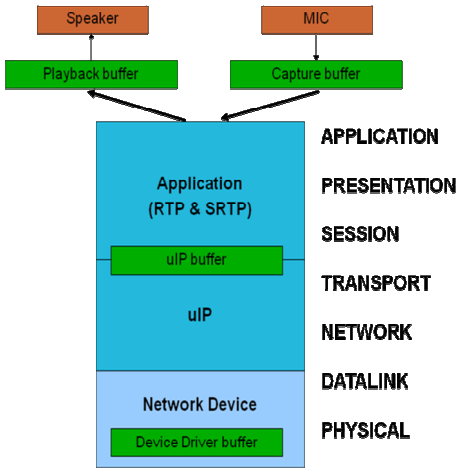


Figure 7. Buffers in uVoIP

The buffers are associated with various states and events. The events are responsible for changing the state of a buffer which includes

1. Net in start 2. Net in complete 3. Capture complete
4. Playback complete 5. Netout complete.

The 6 states associated with the buffers are

1. Free 2. Capture 3. Playback 4. Netin
5. Output 6. Output Pending 7. Playback Pending.

We built a custom discrete event simulator to investigate the minimum number of buffers required to process, play, and send voice data without any packet loss. For discrete-event simulation, operation of the system is modeled as generation and processing of a sequence of events.

The state of buffers during the occurrence of various events such as capture complete, net in complete, net in start, net out complete, and playback complete is described in Figure 8-12 respectively.

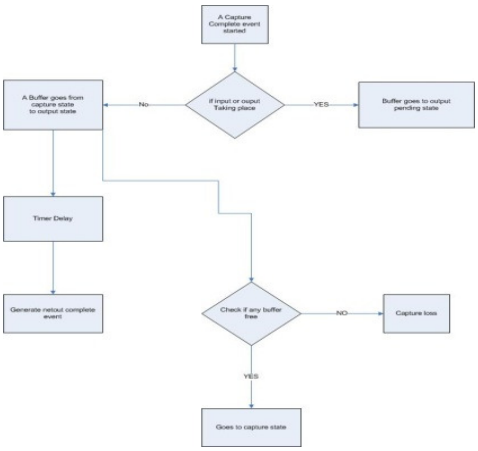


Figure 8. Capture complete event

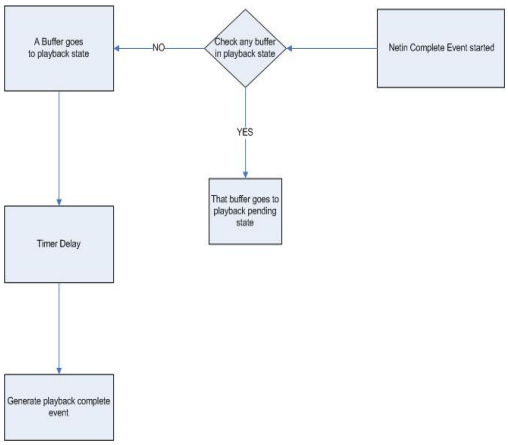


Figure 9. Net in complete event

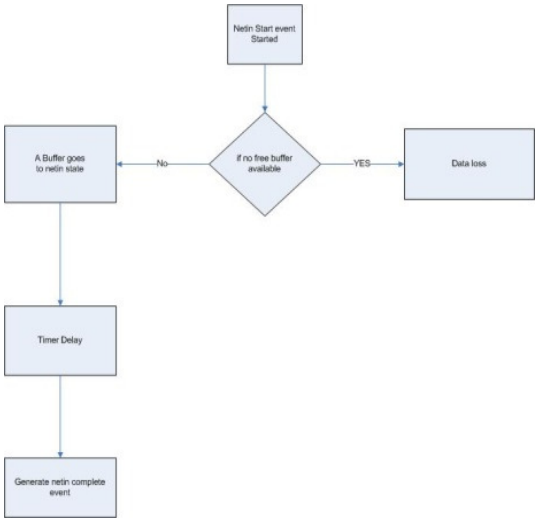


Figure 10. Net in start event

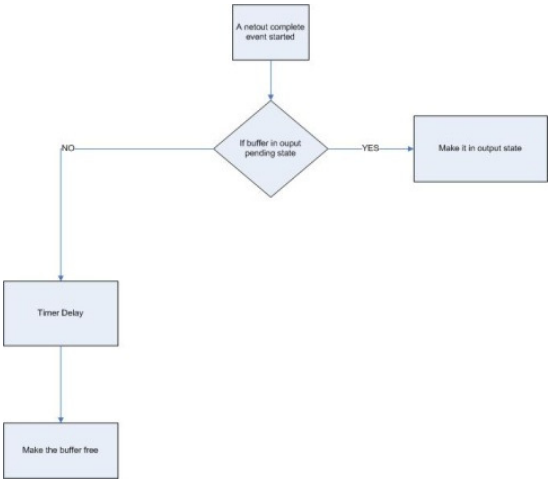


Figure 11. Net out complete event

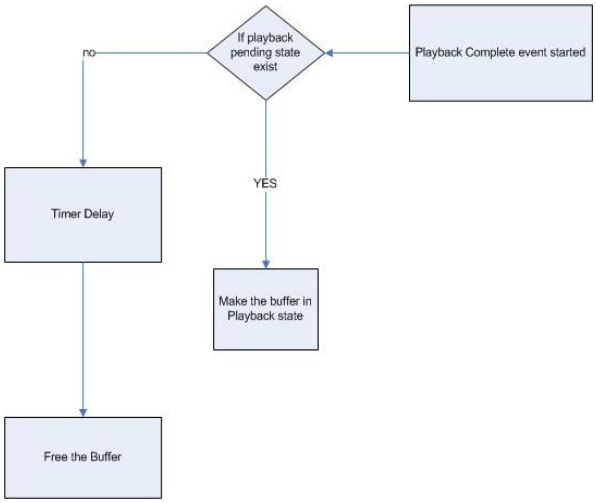


Figure 12. Playback complete event

It may be noted that buffers are dynamically allocated and buffer flipping is used to reduce number of buffer operations.

## 7. CODE ARCHITECTURE

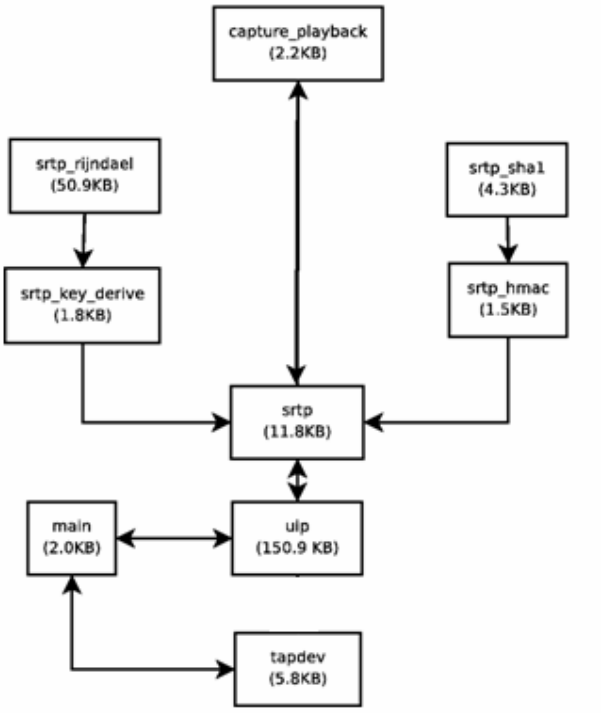


Figure 13. Code Design

Figure 13 shows the architecture of the uVoIP stack code. A brief description of the different modules follows.

*Capture_playback* playbacks the received packets to the Loud speaker and continuously capture the voice data from the Microphone.

*srtp_rijndael* implements AES-CTR algorithm.

*srtp_key_derive* derives the keys needed for encryption, authentication and session.

*srtp_sha1* implements Hash Function.

*srtp_hmac* produces Authentication Tag.

*srtp* creates SRTP packet for captured data. For received packets if authentication is successful it decrypts the data.

*uip* implements all TCP/IP layer functions

*tapdev* implements Ethernet layer functions.

*main* initializes all buffers. This module periodically checks to see whether a packet is received. If a packet is received it calls uip to process the data. It also periodically checks whether a packet is completely captured. If a packet is captured it passes the data down the stack (RTP, SRTP layers), and then calls uip functions to output the data.

## 8. RESULTS

A summary of our experimental results are given below.

1. Successfully implemented uVoIP stack and sent the encrypted and authenticated captured sound data from the sound card of one system via tunnel interface and played back in other system of different network after successful authentication and decryption.
2. After successful one way communication, we successfully did two way communications in the same manner as above.
3. Proved using discrete-event simulation that minimum number of buffers that can be used with minimum data loss is three and with no data loss is four.

### 8.1. Simulation Results

We investigated the performance of the system using three buffers and four buffers and measured the packet loss for variable jitter values and a latency of 150ms(maximum Ethernet latency). The packet loss obtained for 3 buffers is plotted in Fig. 14. With the usage of 4 buffers, we incurred no packet loss.

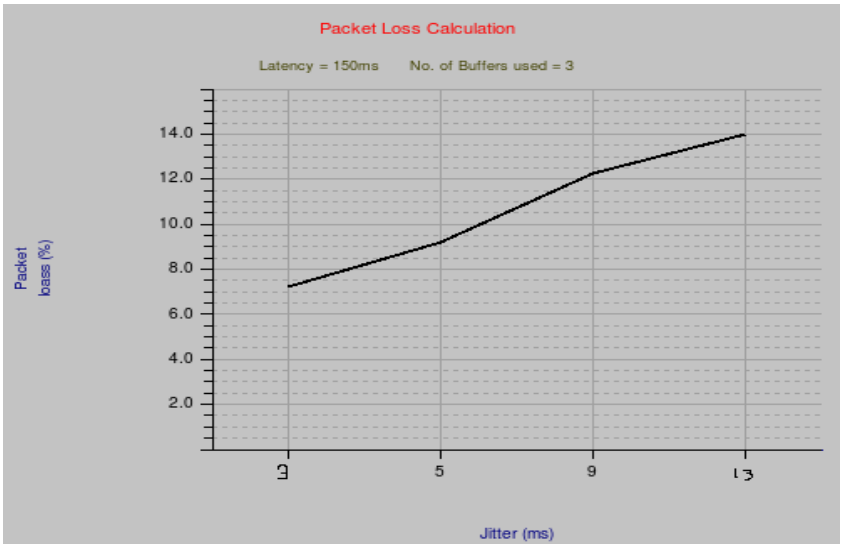


Figure 14. Packet loss for usage of three buffers

## 9. CONCLUSIONS

In our paper, we have implemented an optimized uVoIP stack which requires only four buffers across all layers and also optimizes the number of buffer operations. We have verified our cross-layer optimization technique by building a discrete event simulator and verifying the simulation data. The uVoIP stack has been designed in such a way that it can be easily ported to any different embedded platforms and imposes minimal requirements for clock speed, data memory, and program memory. We will be continuing this work by analyzing the performance of the stack in various embedded environments.

## ACKNOWLEDGEMENTS

We offer humble salutations to our Guru, Shri. Mata Amritanandamayi Devi, who inspires and guides our good thoughts and actions.

## REFERENCES

[1] H. Schulzrinne,R. Frederick,V. Jacobson, "The Real-time Transport Protocol," RFC 1889, Internet Engineering Task Force, Jan 1996.

[2] D. McGrew, E. Carrara, K. Norrman, Secure, "The Secure Real-time Transport Protocol," RFC 3711, Internet Engineering Task Force, Mar 2004.

[3] Adam Dunkels, "The uIP Embedded TCP/IP Stack," Swedish Institute of Computer Science, 1995.

[4] R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload," RFC 3686, Internet Engineering Task Force, Jan 2004.

[5] H. Krawczyk, M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication ," RFC 2104, Internet Engineering Task Force, Feb 1997.

[6] (2009) The Tunnel interface documentation website.[Online]. Available: http://www.mjmwired.net/kernel/Documentation/networking/tuntap.txt .