SIMPLIFIED PROTOCOL CAPTURE (SIMPCAP)

Michael W. Corley*, Michael W. Weir†, Kenric Nelson†, Andrew J. Karam‡

Abstract--Standard packet capture architectures have no inherent support for protocol decoding. Client systems are required to handle all decoding and maintenance of protocol handling constructs in a proprietary fashion. Resultant system architectures are often not optimized and difficult to expand upon, particularly for defining and implementing new and unhandled protocols. This paper describes a new protocol decoding system called Simplified Protocol Capture (SIMPCAP). The system, developed for legacy use with LIBPCAP, constitutes an optimized high-level library architecture that automates protocol decoding and maintains protocol definition knowledge constructs globally. The SIMPCAP framework incorporates a high level API (application programming interface) for convenient and flexible access to protocol field state.

Index Terms--network packet capture, protocol demultiplexing, intrusion detection

I. INTRODUCTION

Most of today's network interface and analysis tools, such as tcpdump [1], SNORT [2], Ethereal [5], etc. are based on LIBPCAP [4], a widely used standard packet capture library that was developed for use with the Berkeley Packet Filter (BPF) [3] kernel device. Essentially, the BPF is an extension to the OS kernel enabling low-level communication between the operating system and a network interface adapter. Traditionally, network analysis tools (architectures) are composed of three major components: the BPF, LIBPCAP, and the client system as depicted in Fig. 1. The major limitation exposed in this model exists between LIBPCAP and the client system. As shown, no inherent functionality exists to aid the client system with the overhead of decoding and interpreting the raw byte encoded sequence that comprises a network packet. Programmers are therefore left to deal with this in a completely proprietary fashion. This is clearly a process that adds significant design and development overhead and often requires extensive programmer expertise. Even worse, this is a development step that is repeated many times and in many different ways depending on issues such as time constraints and programmer experience. More often than not, users and maintainers of such systems are network analysts who do not possess the necessary programming background to be effective maintainers as such. This results in a boundary that often keeps an analyst from exploring new ideas and directions.

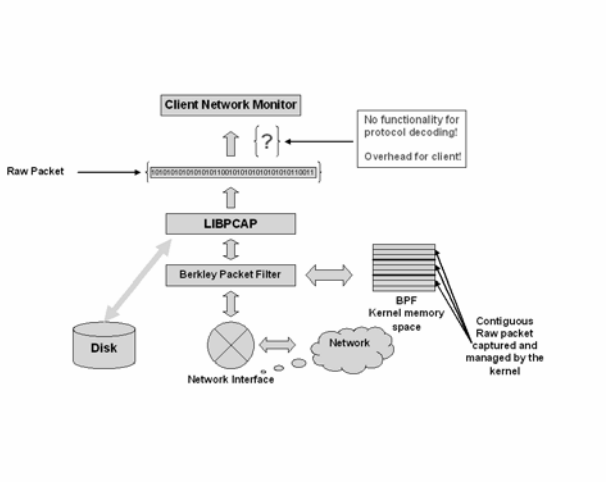

Fig 1. Standard Packet Capture Architecture: LIBPCAP (WINPCAP)

The system we have developed to address these issues is called Simplified Protocol Capture (SIMPCAP). It is unique to the network analysis and software development communities for the following reasons. First, it is designed for legacy use with LIBPCAP, so that no alteration to current approaches is necessary; any code that is written with SIMPCAP can run side by side with any other code written to incorporate the LIBPCAP library. Second, decoding constructs are optimized and automated, requiring only minimal programming maintenance; it is designed with the analyst in mind. Additionally, the architecture is designed for rapid introduction of new and unhandled protocols into the system; a few simple script-like changes are all that is necessary to expand the functionality of SIMPCAP to provide new protocol decodes and data to the client program. This is essential to non-programmer analysts who are often under heavy time constraints. Protocol definition and handling details are statically kept within the system framework. This allows clients to efficiently share a common protocol definition knowledge base and provides application independent protocol decoding functionality. In other words, protocol decoding constructs are portable among all clients and in many development environments. Finally, the system exports a highly expressive C-language API (application programming interface) that maintains convenient robust access to decoded protocol state, as well as the inherited

*Lockheed-Martin, 32 Brooks Rd., Rome, NY 13441
†*SI International*, 1300B Floyd Ave., Rome, NY 13441
‡AFRL/IFGB, 525 Brooks Rd, Rome NY, 13441
*Correspondence: (315) 330-3746, mike-corley@acs-inc.com

LIBPCAP capture functionality. The constructs of the API are particularly convenient for development of next generation intrusion detection and anomaly detection systems that encompass a great deal of statistical analysis on protocol header data.

Other researchers have recognized the deficiency in relying on client software to decode network protocols and have developed software applications that address aspects of the mentioned shortcomings. These include, PYLIBPCAP [8], which essentially comprises a set of python scripts that provide packet decoding constructs and sufficient client access to decoded protocol content. The LIBNET [9] system is a high-level API for constructing and injecting LIBPCAP packets. It exports a portable interface for low level packet shaping, and shields the client from tedious low level details such as those associated with protocol handling and demultiplexing. The System for Modular Analysis and Continuous Queries [10] provides a Dynamic Type System which uses polymorphic techniques to provide flexibility in extracting different packet types. The remaining sections of this paper are organized in the following manner: Section 2 provides an overview of the Berkeley packet Filter (BPF) kernel device and the LIBPCAP capture system, followed by a brief description of the limitations of the client architecture model. Sections 3 and 4, respectively entail an overview of the SIMPCAP system and a detailed explanation of the system architecture. Section 5 briefly describes the SIMPCAP role in a current research initiative called the Second Stage Intrusion Monitor (SSIM). Section 6 summarizes SIMPCAP and provides a view of additional future directions for the package.

II. STANDARD PACKET CAPTURE OVERVIEW

A. BERKELEY PACKET FILTER (BPF) OVERVIEW

As mentioned above, traditional network capture architectures are based on the LIBPCAP library and the Berkeley Packet Filter (BPF) code. The BPF is the low level capture device that provides raw access to data link layers in a protocol-independent fashion. It is essentially a set of low level routines wrapped by C-language function calls to provide the client programmer with higher level system functionality for opening and managing optimized packet capture sessions. The most attractive feature of the BPF is the filter machine on which its name is based. This comprises an expressive instruction set for defining filters to instruct the capture device to retrieve all packets that match a user defined filter (template). See the example illustration in Fig 2 [11]. Although the BPF is a highly efficient raw capture environment, it is generally not suitable for developing network capture applications unless unusually low-level access is a requirement. Managing a BPF capture session can be tedious in that it often involves a significant degree of interaction between user- and kernel-level processes. The client retrieves packets by performing reads directly into the kernel buffer space. In addition, the BPF does not export a standard capture file format; there is no functionality for managing the raw streams and providing a structured output that can be conveniently stored or processed. Live capture sessions can not be conveniently stored or ported to different tools for analysis.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

Fig. 2. Example BPF Filter that accepts only IP packets between host 128.3.112.5 and 128.3.112.35

B. LIBPCAP OVERVIEW

LIBPCAP [4] is a high-level C language library that extends the BPF library constructs. In particular, it provides convenient system functions for managing live and offline capture sessions, and for interacting with and retrieving individual packets from the BPF kernel space. The LIBPCAP capture file format provides a standard and efficient construct for handling offline capture sessions. Captures files are portable to a broad range of operating system platforms and provides convenient accessibility to any tool that conforms to the standard. LIBPCAP further extends the functionality of the BPF filter constructs in allowing programmers to specify filters in a higher-level syntax (such as that used by tcpdump [1]) as shown in Fig. 3. In particular, LIBPCAP implements functions to compile the filter syntax to the native BPF instruction form.

```
tcp[((tcp[12] & 0xf0) / 4):4] = 0x51554954
```

Fig. 3. Example LIBPCAP filter to match the string ‘Quit’ in the TCP header.

The LIBPCAP file format coupled with the BPF filter accessibility forms the heart of the functionality of many network analysis tools in use today, including tcpdump and Ethereal. Although LIBPCAP does provide convenient access to packet data, there is no inherent functionality for decoding the protocol content encoded in the raw packet. Until the writing of this paper, this has been the full responsibility of the client, which is clearly depicted in the upper half of Fig. 1. In fact, proprietary protocol decoding, interpretation of protocol content, and matching of such content to some form of template has been the basic mindset of many systems used for intrusion detection.

C. LIMITATIONS OF THE CLIENT ARCHITECTURE MODEL

The basic operation of the LIBPCAP architecture is to interact with the BPF kernel and provide seamless access to individualized packets. The packet contents are, however, raw byte encoded. Currently, there is no intrinsic support to provide convenient decoding functionality. Therefore, client architectures are proprietary and often difficult to expand upon, particularly for defining and processing new and unhandled protocols. Since the applications are proprietary, their extension to handle new and different protocol information is not guaranteed and can require extensive source code modification. Code level modifications often lead to serious architectural integrity risks. Insufficient programmer expertise or too many disparate programmer modifications can lead to loss of robustness, erroneous runtime errors, degraded performance and even system compromises such as buffer overflows. In terms of the sheer volume of data that may need to be analyzed by a client application, the framework for which the protocol content memory space is utilized by a client should be standardized and isolated from the overall system architecture; there is no such mechanism within the LIBPCAP and BPF arrangement.

III. SIMPLIFIED PACKET CAPTURE (SIMPCAP) OVERVIEW

SIMPCAP is a flexible protocol decoding system matched to a rich client-side API and designed for follow-on use with the standard packet capture system LIBPCAP. Fig. 4 compares the framework for packet capture and client interaction between the typical BPF/LIBPCAP approach (right side) and the modified SIMPCAP architecture (left side). Recall from the previous discussion, LIBPCAP essentially delivers a raw byte encoded packet to the client. The primary objective of SIMPCAP is to isolate the complexities of defining and handling various protocols and to shield the client from such details, for the purposes of obtaining smaller, more robust and portable client designs.

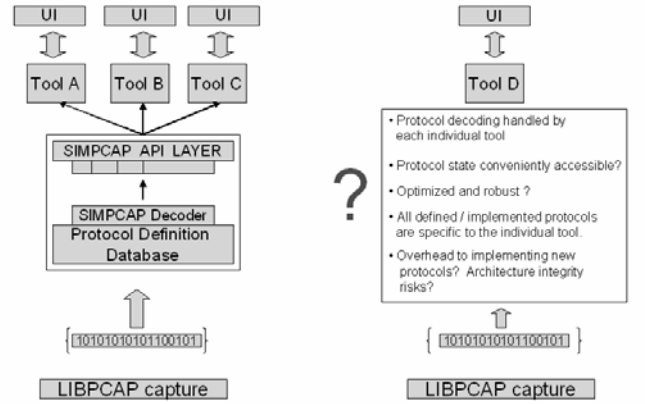

Fig. 4. SIMPCAP code layer between LIBPCAP and client.

This effectively alleviates the client from managing protocol decoding overhead, and allows for direct interaction with protocol field data, rather than the conventional tedious interaction with raw packets. Note that SIMPCAP still maintains the full capture functionality as provided by LIBPCAP and the BPF; the "client" could be a simple pass-through application that just hands off the raw data directly to traditional packet processing applications.

SIMPCAP also standardizes the procedure to implement new or modified protocols as illustrated in Fig. 5 (and covered in detail in the following sections). This standardized approach helps ensure architectural integrity and significantly reduces requirements for software programmer expertise, as is often needed for maintaining or modifying conventional applications. A more important aspect of this approach is the ability to share a global protocol knowledge database. The Protocol Definition Database (PDD), illustrated in the left-middle section of Fig. 4, contains all protocol definitions in the system. All clients written to use SIMPCAP can use the most current PDD available and immediately have access to the new protocol definitions built up by the community.

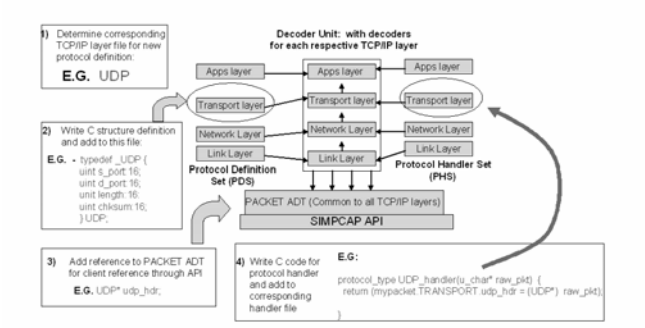

Fig. 5. Adding new protocols to SIMPCAP.

To provide maximal flexibility for client-side applications, SIMPCAP provides access to protocol field state through a C language API as shown functionally in Fig. 6. It enables both experienced and beginner programmers to take immediate advantage of the expressive constructs of the programming environment and combine them with SIMPCAP to achieve the optimized and powerful development requirements of next generation intrusion detection and anomaly detection systems. For instance, consider an intrusion detection system that relies more on heuristics and complex statistical analysis of protocol header data for raising alerts, rather than the conventional system approach that involves matching packet content to some sort of template or rule set to trigger an event. The client-side programmer can work directly with the field characteristics of the packets, and let SIMPCAP figure out the lower-level steps necessary to gather the information and provide it to the client application.

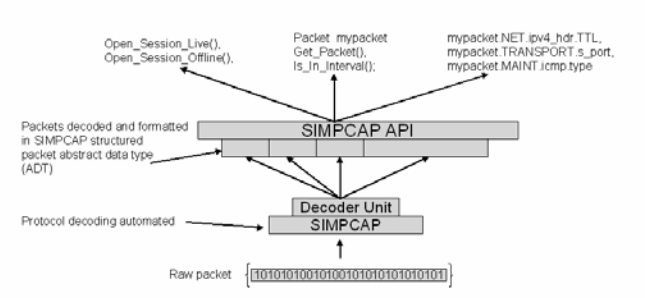

Fig. 6. Application Programming Interface (API) overview.

SIMPCAP uses efficient memory storage/retrieval techniques to maintain a workable user environment that does not scale linearly with the size of the capture file processed. As packets are processed through SIMPCAP, the memory space is cleared and re-used as soon as the decoded information is passed on to the client application.

It is important to re-emphasize that the SIMPCAP framework constitutes a wrapper for the LIBPCAP architecture. Therefore, the raw encoded packet data, as well as the all of the LIBPCAP and BPF capture system functionality is available through the system API layer. Not only does this maintain maximal flexibility, it provides a convenient environment to transition SIMPCAP constructs into currently available tools. The motivation here would naturally be to enhance or replace the system architecture for expanded flexibility and functionality to meet next generation requirements (e.g. interpretation / handling for IPv6 [6]), as well as provide the foundation for a much richer set of protocol definitions that can be shared across disparate client applications while maintaining compatibility with current tools. In essence, this provides a means to simplify the application architecture for new tools, keep the old tools, and begin building tools that address the analysis of packet data and not the processing of the packets themselves.

IV. SIMPCAP ARCHITECTURE

The SIMPCAP architecture model is designed to efficiently decode packets by individual layers corresponding (initially) to the TCP/IP [6] protocol suite. The framework is comprised of four primary entities as depicted in Fig 7. They are as follows: an automated protocol decoding unit (PDU), the common packet abstract data type (ADT) that feeds the decoder unit from the SIMPCAP API, a protocol definition set (PDS) that defines the entities that are available to the decoder unit, and a protocol handler set (PHS) that determines the disposition of the packet entities that are processed by the decoder unit. The PDS is comprised of four layer-specific protocol decoder units.

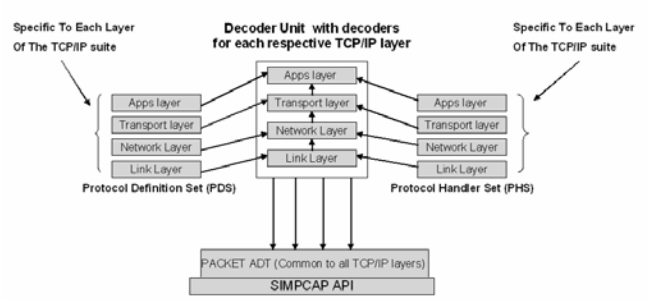

Fig. 7. Detailed SIMPCAP architecture overview.

Each is designed to work in unison to demultiplex each layer of the raw byte encoded packet. The PDU will be further discussed in next section. Each PDS/PHS set contains four distinct source code files that correspond to exactly one layer of the TCP/IP protocol suite as illustrated. Specifically, protocol definitions and handler functions are individually defined for the link layer, network layer, transport layer, and application layer. An example implementation of the transport layer component of a new system protocol is illustrated in Fig. 5. First, an appropriate name is chosen for the new protocol definition (here UDP). The C-code protocol definition (step 2) is built and added to the appropriate layer PDS file (here TRANSPORT), becoming part of the protocol definition set. Then, a pointer reference to the protocol type definition is added to the appropriate section of the packet ADT for access via the API. Finally, a handler for the protocol is included in the corresponding source code file of the PHS. The PDU, using the PHS, is responsible for forwarding protocol references to the packet ADT as discussed in the following section.

```
typedef struct _PACKET_ADT
{
  struct pcap_pkthdr        pcap_hdr;
      /* reference to PCAP header,  (timestamp, etc) */
  const u_char*             raw_packet;
      /* reference to the raw packet as delivered by libpcap device */
  int                       frame_type;
      /* 1 = ETHERNET (RFC 894)   2 = IEEE 802.3 (RFC 1024) */
  LINK_LAYER                LINK;
      /* access to link layer protocol break down */
  NETWORK_LAYER     NET;
      /* access to network layer protocol break down*/
  TRANSPORT_LAYER TRANSPORT;
      /* access to transport layer protocol break down */
  MAINTENCE                 MAINT;
      /* access to maintence protocol (ICMP, IGMP) break down */
} PACKET;
```

Fig. 8. Packet ADT (abstract data type) – the primary user interface to the protocol field state. The layered protocol boundaries are segregated and no function call is needed to retrieve the field state.

As illustrated in Fig. 7 the packet ADT is common to all layer specific entities for the PDU, PDS and PHS. The packet ADT construct is sub structured to contain sections

corresponding to each layer that comprises a TCP/IP packet. Fig. 8 and Fig. 9 show code snippets of the ADT format, structure and usage.

```
/* reference the LINK layer frame types:  ethernet and IEEE 802.3 */
typedef struct _LINKLAYER
{
 ETHER_HDR*            ether_hdr;
       /* reference to standard ethernet header (RFC 894)      */
 ETHER_802_3_HDR*   ieee_802_3_hdr;
       /* reference to IEEE 802.3 (RFC 1042)  */
 ARP_HDR*              arp_hdr;
       /* reference to arp / rarp */
 const u_char*         payload;
       /* reference to the frame payload */
 unsigned short        emb_p_type;
       /* the type of link layer protocol present, IP, RARP, ARP, etc)
       */
} LINK_LAYER;
```

Fig. 9 Link Layer structure with references to the Ethernet and IEEE 802.3 frame types.

A. THE PROTOCOL DECODER UNIT (PDU)

The system performs demultiplexing by incorporating a decoder unit for each respective TCP/IP layer into a common protocol decoder unit (PDU). The process begins naturally with the link layer-specific constructs of the PDU. Using the PDS and the PHS, the PDU invokes the appropriate protocol hander which strips the link layer protocol from the raw byte sequence. A reference is then forwarded to the corresponding portion of the common packet ADT and the embedded link layer protocol is handed off to the network layer decoding constructs. This process is repeated until the entire raw byte encoded packet is fully decoded as depicted in Fig. 10. When the PDU execution is complete, the fully decoded packet contents are accessible through the respective sections of the API.

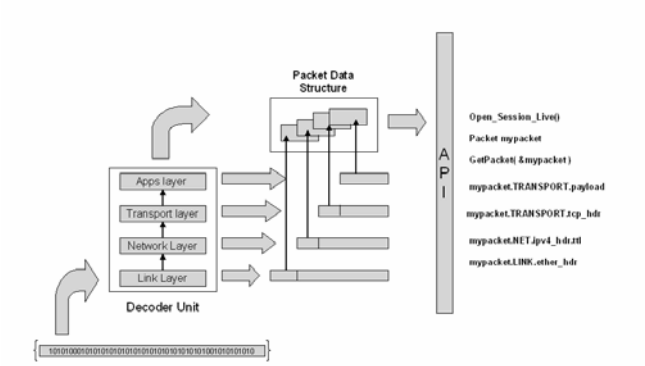

Fig. 10. Decoder Execution Flow.

The PDU unit employs an innovative technique to carry out the decoding process. Each layer-specific decoder is implemented with a vector lookup table (VLT). Each layer-specific VLT functions in essentially the same manner. All index positions of the VLT are associated with the corresponding protocol identifier for that index. Take the link layer VLT for instance, the internet protocol version 4 (IPv4) [6] is an embedded link layer protocol. It is identified by the 16-bit Ethernet frame type code 080016 = 204810 [6]. Therefore, the index position of 204810 in the link layer VLT contains the memory address of the protocol handler function for IPv4. Explicit reference to this vector location would therefore resolve to the invocation of the handler function for IPv4, and header contents would be stripped from the otherwise raw byte sequence. The process begins as the raw packet enters the link a layer decoder of the PDU (refer to Fig. 10). Although the system is designed with consideration of handling multiple frame types, the current implementation is limited to Ethernet. The 16-bit Ethernet type code is used as an index into the link layer VLT. This resolves to the appropriate handler invocation for the embedded link layer protocol encapsulated by the Ethernet frame. If the embedded protocol is one that defines further embedding (such as IPv4 or IPv6 [6]) then the appropriate protocol field is used as index into the network layer VLT. In the case of IPv4, the 8-bit protocol field is used as index to the network layer VLT. This in turn resolves to the appropriate invocation of the handler for whichever embedded network layer protocol is being carried (e.g. TCP, UDP, ICMP, etc.) [6]. As mentioned previously, each handler must include all code necessary to forward a reference to the appropriate section of the packet ADT. This technique is attractive in that the PDU execution is begun by the initial reference to the link layer VLT. All other components of the decoder are invoked in serial automated fashion as the protocols for each layer are demultiplexed.

V. CURRENT APPLICATIONS

The SIMPCAP framework provides the network packet interface component for a research and development project currently being conducted by the Information Directorate at Air Force Research Laboratory, Rome Research Site, Defensive Information Warfare Branch (AFRL/IFGB). The Second Stage Intrusion Monitor (SSIM) is an initiative to provide innovative statistical and information theoretic techniques for next generation intrusion detection and anomaly detection systems. The basis of the SSIM architecture is the SIMPCAP code, which allows the client application programmer to focus on the selection of packet features and the analysis tools to apply to those features to determine the existence of anomalous behavior. As depicted in basic SSIM architecture diagram of Figure 11, SIMPCAP is central to all applications (clients), conveniently alleviating protocol decoding and handling overhead. Within this architecture, each application is a component of the system that builds specific statistics about the underlying traffic content. The statistics are combined to build heuristics, visualize correlation among disparate events, and provide baseline input for the information theoretic processors.

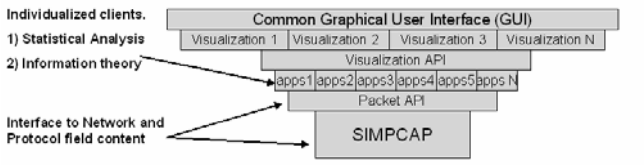

Fig. 11 Second Stage Intrusion Monitor (SSIM) Overview

Client-side applications being designed for the SSIM include information theoretic processors comprised of entropy measures as devised by C. Tsallis and C. Shannon [7]. These techniques provide a baseline for modeling systems that contain long range correlation, long term memory, or contain inherently self similar constructs. SIMPCAP provides an efficient engine to deliver packet characteristics and information to these components for modeling network traffic anomalies.

The applications are powerful in that they are not limited by individual network interfacing constructs. Although the interface to the raw network content is maintained through SIMPCAP, applications as such need only provide explicit references to the protocol field contents.

VI. CONCLUSION AND FUTURE DIRECTIONS

In this paper we have presented a flexible and extensible protocol decoding engine and client-side API called SIMPCAP (Simplified Protocol Capture). The system is designed as a convenient and efficient interface between LIBPCAP and client applications. This provides the necessary constructs to isolate and remove protocol decoding requirements from the client application. The decoding aspects are automated, optimized, and easily extensible for unhandled protocols, providing a convenient environment for non-programmer analysts to explore new ideas and directions. The system exports a flexible and highly expressive C-language API enabling users to efficiently combine a development-rich environment with network protocol analysis. It also allows for a standardized approach to integrate community-developed protocol decoders into a shareable processing environment. It currently runs on both Linux and Windows platforms. Two promising future directions for this project include a virtual file representation for relating multiple trace (capture) files based on user selectable criteria, and a packet capture file processor that is multi-threaded and can be used independently from the LIBPCAP library, allowing parallel processing of large packet captures.

The virtual file representation will enable SIMPCAP to deliver a packet state that is representative of one or more related trace files through a seamless session. Analysts typically perform some sort of file manipulation for extended capability or added convenience in normal processing. There is often the desire to physically concatenate multiple capture files, to extract particular regions of files that are suspected to contain certain content, or to view and analyze portions of multiple files comparatively. Currently, a suite of open source data manipulation tools are used to carry out these basic functions, with the real data analysis being done in the analyst's head. As data volumes get larger, and their content more disparate, such functionality becomes more complex and even impractical. A virtual file representation, however, constitutes a preprocessing stage that enables analysts to index one or more capture files conveniently, based on user criteria, and implement visualizations that provide meaningful output. This extension to SIMPCAP will enable the client programmer to open and decode virtual files in much the same fashion as opening LIBPCAP trace files for post-mortem analysis.

The multi-threaded packet-processing approach can be used to reduce the amount of dead time that analysts have to experience when processing large capture files. Rather than use a linear LIBPCAP session to step through each packet in serial succession, the SIMPCAP engine and API (when combined with an appropriate metadata file) can provide a structured binary data set that can be operated on directly and parsed to multiple threads for rapid calculation and collection of information about the capture.

VII. REFERENCES

[1] V. Jacobson, C. Leres, and S. McCanne, `tcpdump`, Available via anonymous ftp at ftp://ftp.ee.lbl.gov.

[2] M. Roesch, Snort – Lightweight intrusion detection for networks, Proceedings of the 13th Systems Administration Conference., USENIX, 1999.

[3] S. McCanne, and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Proceedings of the 1993 Winter USENIX Technical Conference,San Diego, CA.

[4] S. McCanne, C. Leres and V. Jacobson, `LIBPCAP`, available via anonymous ftp, ftp://ftp.ee.lbl.gov.

[5] Ethereal. What Is Ethereal? http://www.ethereal.com/docs/user-guide/x69.html.

[6] R.W. Stevens, TCP/IP Illustrated Volume 1: The Protocols, Addison Wesley, 1994, pp. 1-34.

[7] K.P. Nelson and P.E. Losiewicz, Application of Tsallis Entropy to Bit-stream Analysis, in: Proceedings of the 8th Canadian Workshop on Information Theory, 2003.

[8] D. Margrave, PYLIBPCAP Project, http://mail.python.org/pipermail/python-announce-list/2001-January/000639.html.

[9] LIBNET, LIBNET Documentation, http://libnet.sourceforge.net/#docs.

[10] M. Fisk, G. Varghese, Agile and Scalable Analysis of Network Events, Proceedings of the second ACM SIGCOMM Workshop on Internet measurment, Marseille, France.

[11] S. McCanne and V. Jacobson, The BPF Manual Reference Pages, http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=bpf.