How to do Network Forensics on GSE Overlay Networks

Anonymous Author(s)*

Abstract

Captured network traffic increased on its importance as a data-source for law enforcement crime investigation because everything is becoming internet connected and a suspect's phone or computer communication might yield crucial evidence. There are many points in the Internet Service Provider's infrastructure where the network traffic might be captured. One of them is satellite connection, DVB-S2, which use Generic Stream Encapsulation (GSE) to carry IP traffic. Current tools for network traffic forensic analysis do not support GSE. In this paper, we describe GSE and how we implemented support for GSE into OUR TOOL.

CCS Concepts • **Applied computing → Network forensics**; • **Networks** → *Network monitoring*; *Network protocols*; Transport protocols; Application layer protocols; • **Social and professional topics** → Computer crime.

Keywords network traffic forensics, generic streaming encapsulation, network forensic and analysis tool

ACM Reference Format:
Anonymous Author(s). 2018. How to do Network Forensics on GSE Overlay Networks. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The digital forensics is becoming a domain of filed operatives employed in Law Enforcement Agencies (LEA) that are tasked to investigate crimes. Their data-source might vary, like seized mobile phones, computers, or other storage devices. Long-running investigation cases use a lawfully intercepted network traffic as a valued data-source [2].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Although the analysis of network communication was not considered the primary area of digital forensics, its importance has increased as most of the devices are Internet-enabled. Performing network forensic analysis requires adequate tool support [13, 14]. A typical network forensics analysis tool provides features that aid an investigator to reveal evidence in network communication [1]. Instead of giving comprehensive information on network protocol details, the forensic tool is expected to provide contents of transmitted files, perform a keyword search, extract user credentials, and more [2, 20].

A network analysis tool, without a solid foundation capable of processing a wide range of network, application, and encapsulation protocols, is usable for only a limited use-case or requires expert knowledge of operators to pre-process the data to suit the tool. The field operatives are experienced criminal investigators but usually not computer experts. Therefore, tools they use need to be straight-forward, provide top-to-bottom analysis, and require as few expert knowledge as possible.

The overlay networks are becoming widely used by Internet Service Providers (ISPs) that are interconnecting various public places, businesses, campuses, or regular home internet connections. Technologies can be fiber-optic, metallic ethernet, 3G, 4G, 5G or satellite connection DVB-S2 that uses GSE to encapsulate IP traffic [6, 8–11].

We chose to implement support for GSE on demand of the Czech LEA, and to demonstrate the extensibility of OUR TOOL to support not only new application protocols but protocols on all network layers, even those that can occur on Link or Application layer — like GSE. LEA officers prefer open-source network forensic and analysis tools (NFATs) [1, 12], even though they might be poorly documented, out-of-date, and even abandoned [13].

1.1 Problem Description

The GSE is nowadays commonly used for data encapsulation on satellite networks. As its name suggests, it is a generic method of encapsulation and can occur on any network layer and that even recursively. The LEAs struggle to perform network forensics on data captured with GSE encapsulation because commonly used tools for network forensics cannot process this encapsulation.

1.2 Contribution and Paper Structure

This paper introduces a GSE from a network forensic point of view. We survey NFATs, and Network Security Monitoring

(NSM) tools in search of overlay network, and mainly GSE decapsulation capabilities. It is important to note that no other tool intended for a high-level network traffic analysis for LEAs do support GSE.

Consequently, we provide a detailed description of OUR TOOL architecture and atop of it, we describe how the GSE is efficiently processed.

This work might be used by advanced network forensic practitioners that write their own single-purpose tools to dissect network communication and analyze it as well as those that use network forensic tools for their daily routines and do not require deep insight into processing techniques.

2 Related Work

Network forensic practitioners commonly use two types of tools — the NSM and the NFAT [13]. This section mainly focuses on tunneling protocols support in related tools and their usability for network forensic investigation conducted by LEA officers.

NSM tools are intended for a high-level insight into the network communication. Such tools are usually fast and scalable; thus can process high volumes of network data on high-speed networks up to hundreds of gigabits. These tools provide information typically from lower layers, i.e., Internet and Transport, and only partially from Application, where they parse only well-known protocols; rarely they support overlay networks. Also, these tools are guided strictly by standards and usually do not include heuristics or more in-depth analysis to extract additional content. They operate online, and most cannot process malformed or incomplete communication. The incomplete communication is a typical case when interception is done on commodity hardware inside ISP infrastructure. Therefore, these tools are used mostly by network operators for measurements, accounting, and incident detection. NSM tools provide the bottom-up approach showing dissected packets and letting the investigator conduct expert analysis.

The most commonly known NSM tool is Wireshark [28] that supports following encapsulation protocols: GSE, GRE, Ayiya, GTPv1, L2TP, SSTP, PPTP, IPIP, IPsec, 6in4, etc. It supports the broadest range of network and application protocols. Wireshark defines an API that can be used to extend its functionality by a new protocol dissector. Note, that it is *the only tool supporting GSE*!

Some NSM tools can be integrated, and more sophisticated analysis can be done programmatically, like TShark [28], TCPDump [25], TCPFlow [27], NfDump [19], Suricata [24] (Teredo, GRE), Zeek [30] (Ayiya, Teredo, GTPv1, GRE), Moloch [17] (GRE) that can analyze live or intercepted communication. They can be parts of scripts that can do one or more tasks, but still can not be compared to NFAT carving and analytical capabilities.

NFAT Our focus is to provide a tool for LEA operatives to extract forensically important information mostly from the application layer of communication. This intent perfectly fits into the category of NFATs that is intended for in-depth traffic analysis, that is mainly performed *offline* on captured communication. NFATs provide the same amount of information as NSM tools but also add extra information extracted from the application layer. They conduct a thoughtful analysis of the traffic and use the extracted data to infer information that helps the investigator. The information is usually provided in a synoptic, easily navigable user interface because NFATs are intended to be used even by field operatives without specialized training.

Popular NFATs are NetworkMiner [18] (GRE, 802.1Q, PPPoE, VXLAN, OpenFlow, SOCKS, MPLS, and EoMPLS), PyFlag [3, 21], XPlico [29] (L2TP, VLAN, PPP), NetIntercept [5]. No NFAT supports GSE as far as we know.

3 OUR TOOL in Depths

In this section, we present OUR TOOL a network analysis desktop application created for Windows platform. We discuss the low-level network traffic processing parts to be able to explain the extension of GSE decapsulation support. The tool is composed of two parts:

- **OUR FRAMEWORK** (backend, details see Sec. 3.1) is network traffic processing engine that provides all kinds of functionality starting from capture file loading, going through traffic processing, extraction and ending with traffic analysis.
- **OUR TOOL** (frontend, details see Figs. 10, 11) is a visualization tool that depends on the backend for processing part, but extending it with analytic capabilities to interpret extracted data.

For a high-level overview of the tool, architecture see Fig. 1. Note, OUR FRAMEWORK is a separate set of .NET assemblies that have no dependency on OUR TOOL and can operate separately. However, the framework does not have any CLI and therefore has to be incorporated into an application. On the other hand, OUR TOOL has a direct dependency on the OUR FRAMEWORK and is compiled with it, e.g., it uses types that are defined in OUR FRAMEWORK.

3.1 OUR FRAMEWORK

OUR FRAMEWORK is the backend and it is responsible for parsing and preparing all information gathered. For instance, it identifies used protocols, to overcome fragmentation (L3) and segmentation (L4). In its current version, it does not support live capture but can process standard input file formats such: *libPCAP, Microsoft Network Monitor cap, and PCAP-ng.*

Link Layer Once an input file is loaded, it is processed frame by frame (L2). The lowest used protocols type (e.g., LINKTYPE_ETHERNET (IEEE 802.3), LINKTYPE_IEEE802_11

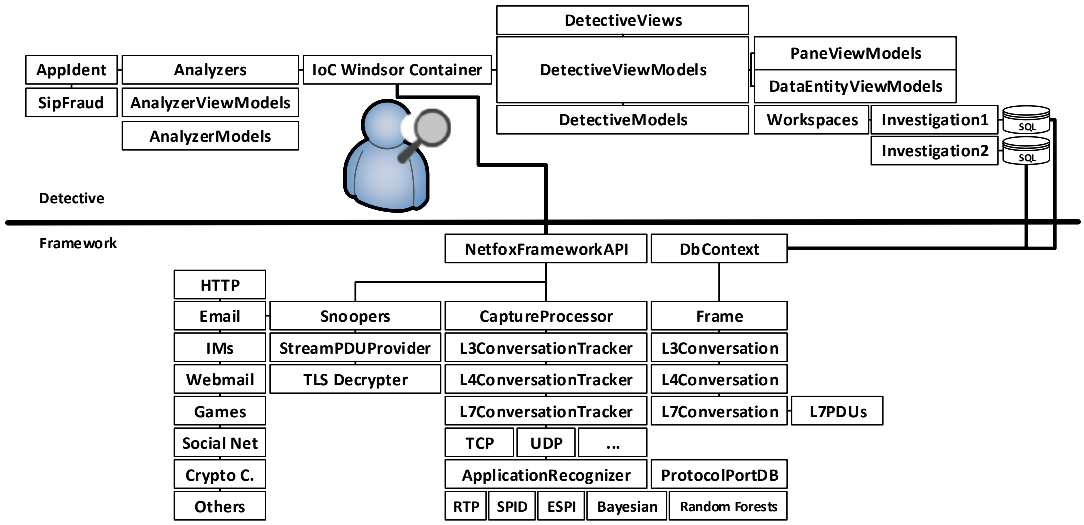

Figure 1. The figure describes the abstraction of OUR TOOL and OUR FRAMEWORK architecture. The upper part of the diagram above the line represents visual parts of the tool. Below the line, components of OUR FRAMEWORK are drawn in a hierarchical view.

(IEEE 802.11), LINKTYPE_PPP, etc.) is stored in the 'pcap_file_header' structure, and we use it to load the first protocol parser. A good overview of the Link-Layer header type values is provided by [26].

Next, we utilize the frame header and its Logical Link Controller header (LLC) where the main field is a unique identifier of the L3 protocol (e.g., IPv4, IPv6).

Note, sometimes it might not be stored in the capture file. Link layer usually does not carry any forensically significant information; thus it is generally omitted and *LINKTYPE_RAW, LINKTYPE_NULL* link layer types are used.

Internet Layer Similarly, both IPv4 and IPv6 contain an identification of an upper layer. (Note, IPv4 names the field 'protocol'; IPv6 names it 'Next Header') which allows us to choose an appropriate L4 parser. As long as the protocol/next header is present, we can parse the communication deterministically, usually up-to (including) the transport layer.

Transport Layer The transport layer carries no information about the subsequent protocol; therefore, the continuing application layer needs to be identified by other means to be correctly processed. We can do this identification using several methods (e.g., port-based classification, deep-packet inspection, probabilistic and statistical methods based on machine learning). Typical encapsulation with protocol examples is presented on Fig. 2.

3.2 Conversation Tracking

This section provides a comparison of ISO/OSI and TCP/IP models with denoted layer names and samples of typical protocols used on particular layers. The logical approach to process network data is to create a *forest of trees* with roots based on identifiers extracted from the lowest layer of the network encapsulation model and continue with upper encapsulation levels. This way, conversations on all levels are created, which also sets boundaries, and specific traffic can be targeted for analysis and information extraction. Each level of encapsulation has its specifics that are to be heated for correct processing.

Besides, each layer has its specifics that need to be taken into account before processing ongoing layer.

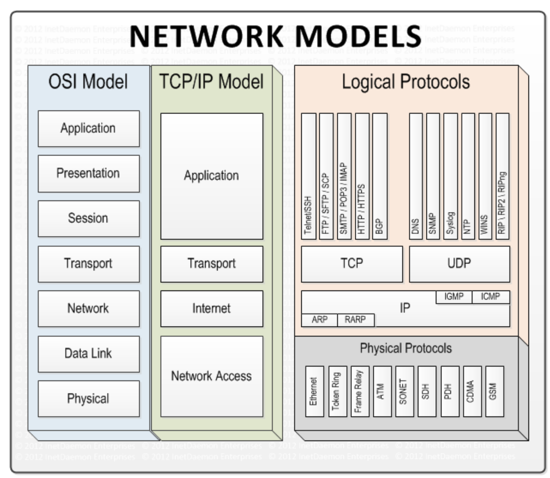

Figure 2.

JP: Draw own figure with supported protocols

This figure provides the comparison of ISO/OSI and TCP/IP models with denoted layer names and samples of typical protocols used on particular layers.

- **IPv4 (L3) fragmentation** can occur, and packets need to be defragmented before further processing. Fragments are identified by *Fragment Offset* and bit *More Fragments* (MF) set in *Flags* field. As long as MF bit is set, defragmentation process has to buffer packets and further process them in bulk, because fragments do not carry headers from upper layers, thus cannot be processed separately and in parallel.
- **TCP (L4) segmentation** occurs regularly. Segments are agnostic to processing mechanisms, carry all required headers and can be processed in a semi-parallel manner. The position of a segment in transmission buffer is defined by the difference of initial sequence number (SYN packet's SEQ) and the particular segment's SEQ.
- **Application messages** are not implicitly denoted because each application protocol has its structure and is not parsed on this level of processing. To obtain at least some level of abstraction, we can deduce boundaries of application messages from the transport layer. E.g., TCP's field *Flags* contains the *PSH* bit that is set when the last segment of a particular application message is created. In other words, when *flush()* is called on network socket which is typically done to notify the kernel that message is to be dispatch right away.

Our unique mechanism of processing network communication [CITATION REMOVED DUE TO DOUBLE-BLIND REVIEW], mainly L4 segregation shown that even malformed or corrupted captures could be used as data-source and carving modules can extract otherwise lost information. We accomplish this during the last processing step, that creates *L7PDUs*, which are the approximations of application messages.

3.3 OUR TOOL Architecture

OUR TOOL was designed to be modular and modules to be inter-operable, but also to work as self-contained libraries to be used by other tools. This way, we have created a framework for network forensics and analytic application supporting forensic investigation.

Fig. 1 describes the decomposition of the tool to small interconnected building blocks/modules. In the bottom part, the architecture of OUR FRAMEWORK processing network communication that is interconnected with OUR TOOL by *OURFrameworkAPI*. This API enables easy incorporation of OUR FRAMEWORK with any additional software that may use it as a platform. Furthermore, this part is divided into two groups, the *execution* and *model* parts.

Execution part, on the left-bottom side of *OURFramework-API*, consists of modules that by their composition ensures polymorphic behavior and extensibility. Each new networking protocol that is to be supported requires the creation of its tracking building block and connection into the processing pipeline. The communication interface between building blocks is defined by their interfaces that buffer inputs and outputs that encapsulates data in models.

Model part consists of blocks below *DbContext*. Models serve as data carriers for parsed, extracted state information, e.g., for L3 conversation it is *source and destination IP address* with a collection of other models representing *Frames*. Models are persisted with *DbContext* and also accessible through it to higher layers.

To ensure fast parallel processing on a single computation node with shared memory, i.e., an application running a single process, we used *Task Parallel Library* (TPL). This approach enables the creation of functional blocks that improve modularity. Each block processes immutable data; thus, all blocks might run in parallel and together create an oriented graph, a Data Flow[1]. The OUR FRAMEWORK combines buffering blocks that interconnect execution blocks to maximize the utilization of resources due to different time complexities of data processing in the functional blocks. Also, this introduces a back-pressure mechanism that is used as memory management to slow down faster blocks that might otherwise overwhelm the system and caused resource depletion and by that, a disk swapping or an application crash.

3.4 Capture File Processing

In OUR FRAMEWORK, capture file processing is initiated by a method call of *AddCapture* in *OURFrameworkAPI*. In current implementation, the tool processes captured traffic in formats *libPCAP*, *PCAP-ng* and *MNM Cap* (Microsoft Network Monitor). Fig. 3 describes a sequence of execution calls and model passing through execution pipeline, a layer by layer to describe logical processing in an abstracted manner.

Modules are designed to ensure concurrent processing thus do process immutable data. Majority of modules also do run in parallel instances to increase a degree of parallelism further. This design also enables with some modifications of processing pipeline to scale up and run the data flow graph in a distributed environment. That is achieved with TPL Data Flow which also enables to change interconnection of execution block to extend the processing of capabilities to process new network encapsulations (tunneling protocols).

The rest of this section describes processing blocks and their interconnections denoted on Fig. 4.

ControllerCaptureProcessor

ControllerCaptureProcessor block is used to oversee captured traffic processing. This module interconnects particular functional and buffering block to a processing pipeline reflecting typical network layered encapsulation. Processing data flow pipeline is created a new for each job. That leads to segregation of data potentially originated from multiple cases and guarantees that no data might be reconstructed into false evidence. The processing has two reading phases.

[1] https://msdn.microsoft.com/cs-cz/library/hh228603(v=vs.110).aspx

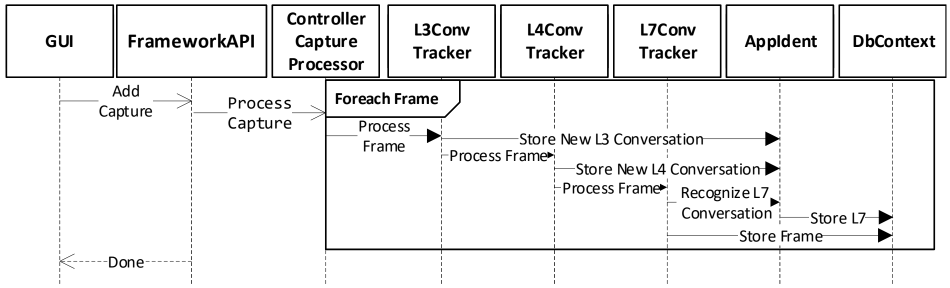

Figure 3. Abstract capture file processing scheme with a sequential passage. Data dependencies between models are omitted. New conversations are stored in relational database triggered by the processing of a first frame belonging to it.

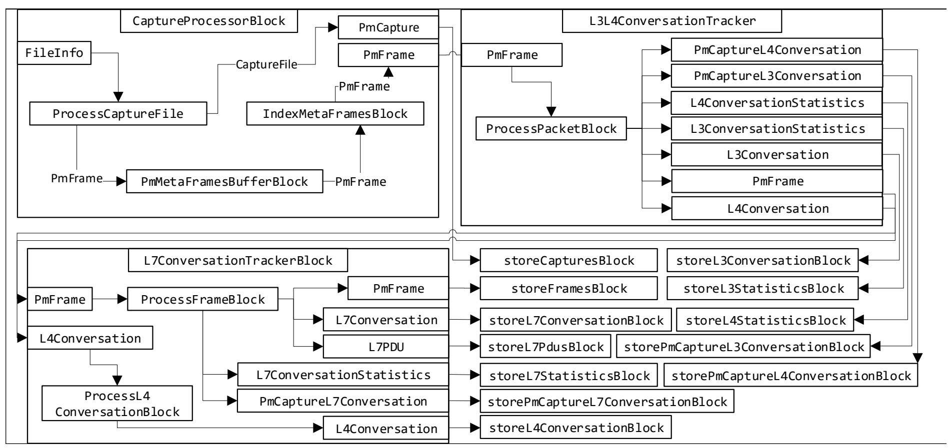

Figure 4. The figure describes the scheme of the functional and buffering block based on TPL Data Flow. This schema describes the decomposition of processing units to perform actions like reading frames from capture files, tracking conversations on L3, L4 levels and furthermore on L7 application layer with the approximation of application messages and application protocol identification.

Firstly, a path to file or files with captured communication is passed to the *CaptureProcessorBlock* that takes care of parsing of particular PCAP file format and retrieving raw frames. The output of this block is *PmCapture* object collection meta information about the capture file and frames encapsulated in objects of *PmFrame*. PmFrame is obtained in the sequential streamed one-way passage of capture file and contains only information about its position in the capture file.

Secondly, additional meta information used in further processing without actual payload is filled in the second read by *IndexMetaFramesBlock*. This segregation is due to a way how frames are stored in various PCAP file formats. Some formats (e.g., MNM) contains a frame table with this meta-information in place and spares the first PCAP read. Execution of *IndexMetaFramesBlock* block, which is a non-blocking read from PCAP file with parsing of (L2), L3, L4 layers, is done with the maximal level of parallelism. Layer 2 might be omitted in case that PCAP is captured without it.

L3L4ConversationTracker

L3L4ConversationTracker takes care of the creation of conversations on particular levels inside the *ProcessPacketBlock*. A PmFrame(s) (packets) with the same IP source and destination address compose a *L3Conversation*. This L4 conversation if furthermore a collection of smaller L4 conversations that composes PmFrame(s) (datagrams) with the same IP source and destination address and TCP or UDP source and destination ports and L4 protocol type (i.e., either UDP or TCP).

In the time when conversations on layer L3 and L4 are created, meta-information in the form of PmFrames is still kept in memory. Because of that, complementary to the conversation creation, conversation statistics are generated as well. Statistics on both levels are updated by data processed from each PmFrame passing through *ProcessPacketBlock*.

Because the processing model in OUR FRAMEWORK is based on IP communication, all non-IP communication is tracked in special aggregation conversations. These conversations have invalid IP addresses as identifiers, i.e., *0.0.0.0* and *[::]* on L3 level, and invalid endpoints on L4, i.e., *0.0.0.0:0* and *[::]:0* as both source and destination. Similarly, L3 conversations containing an unknown transport protocol are aggregated into first L4 conversation with valid IP addresses but invalid transport ports, i.e., 0 port number.

L7ConversationTracker

L7ConversationTracker is a core of our reassembling engine currently supporting TCP and UDP transport protocols. Various TCP heuristics [16] are used to separated IP flow communication, i.e., L4 conversations to finer-grained units based on application session. We call them L7 conversations.

This module processes incoming datagrams in parallel respecting the following scheme. For each newly processed L4 conversation creates a new Task and stores it into a dictionary keyed by an L4 conversation key. All consequently processed datagrams will be forwarded into this task. Tasks run in parallel on multiple cores and are scheduled by the TaskScheduler inside Common Language Runtime (CLR), which makes them much lighter than regular OS threads because they are running on existing threads stored in the ThreadPool. After a task is done or paused, the thread is returned into the ThreadPool, and a new task is immediately executed on it. This way, the overhead is minimal, and parallel processing improves performance rapidly.

Based on the transport protocol type, appropriate reassembler is selected, and the datagram is passed to it for the processing. Reassemblers incorporate heuristics [16] for advanced network traffic processing capable of accurate processing of even malformed, or missing frames.

- **UDP reassembler** uses timeouts to separate consequential UDP sessions. Because of a lack of information from UDP protocol, application messages are created as an ordered sequence of *L7 PDUs*. Each L7 PDU contains only one datagram.
- **TCP reassembler** is more complex and uses properties of TCP protocol like sequence numbers, flags (mainly SYN, FIN, RST, PSH) in combination with timeouts. Based on TCP properties, approximations of application messages are created in the form of the ordered sequence of *L7 PDUs*. Each L7 PDU contains one or more datagrams composing the application message.

TCP Reassembler This solves an issue with the ambiguity of L4 conversations captured in one or many simultaneously processed captures. Typically this happens when static ports are used at server and client side. In a case when a packet loss corrupts capture, it may happen that multiple TCP sessions would be merged into one because from a network point of view, communication would match the regular schema. A TCP finite state machine would process this merged communication and report missing data but would lack further information. That would result in ambiguity in determination who was communicating, whether there were one or more identities involved.

Both reassemblers (TCP and UDP) produce *L7 Conversations* that contain collections of *data* and *non-data* frames. Non-data frames are frames without payloads that serve for signaling purposes like TCP ACKs, or frames with payloads that are malformed, or retransmitted. These frames do not participate in final stream creation, but their presence is either way recorded for auxiliary forensic intents.

L7PDUs Data frames are stored inside L7 PDUs. One L7 PDU represents a data stream that is an approximation of an application message. An application message is considered to be a sequence of datagrams containing one user action, e.g., the user sends a message on online chat, or an email, or downloads a picture, etc. *Although, one application message can span across multiple L7 PDUs, scarcely, one L7 PDU would contain multiple application messages.* This also serves as a check-pointing mechanism in case that module extracting data from the application protocol is unable to parse the data stream due to corruption or unknown content correctly. We observed that this happens a lot when proprietary application protocols are involved because of their volatile nature and closed specification.

Storage Blocks

Storage blocks are used to assure asynchronous persistence of gathered meta-information in the form of outputs of all functional blocks, i.e., *L3, L4, L7 Conversations with statistics, L7 PDUs and Frames*. Data is stored in SQL database in bulk operations to achieve higher performance with a cost of delay introduced with buffering. Buffering and database storing operations run in separate tasks. This way, both services run in parallel and do not block one-another under ideal circumstances. Storage buffering is highly memory consumptive; therefore, in case that database is slower then processing, back-pressure mechanism protects processing pipeline against memory deprivation lowering its performance.

Bulk insert operations increase performance, but at the same time increases the complexity of processing logic. The first limitation is loosed database consistency because it is not guaranteed that all dependencies are stored before an object that depends on them. In other words, a *L4 conversation* has a dependency on a *L3 conversation* that it belongs to. But they both are stored separately in different storage blocks, thus, in a given point in time it may happen that only *L4 conversation* is stored and *L3 conversation* is not yet present in the database. For this reason, unchecked retrieve of meta-information form database may occur after all captured communication is processed. In the case when meta-data is required sooner, referential integrity needs to be validated explicitly because bulk operations bypass foreign-key constraints.

For bulk operations to be possible, all foreign keys need to be known before a first item is stored in a database. To achieve this, GUIDs [15] provided by a system call are used as object identifiers and guaranteed to be unique. This approach also spares one database trip to retrieve otherwise database generated IDs. Because ID generated during meta-data object construction and this same object is passed through the processing pipeline, all layers that need to satisfy dependencies can do so. That is also the reason why *storage blocks* are connected in the processing pipeline last, see Fig. 4, and all objects created in lower layers are passed to upper ones.

The processing is finished when all *storage blocks* are completed. That signals that all data are stored in the database successfully, and consistency is acquired. The control is returned back through *ControllerCaptureProcessor* to the application code that called *OURFrameworkAPI*. There are no objects directly transferred between the application and framework. Thus the *DbContext*, i.e., the connection point to the database, has to be used to retrieve the data.

Currently, there are two persistence providers supported. The first is the SQL server adapter that is the default for the *Entity Framework* that provides full-fledged capabilities. The second option is mostly for ad-hoc, swift investigation or development that stores all data in memory. We have implemented this in-memory provider to be fully interoperable with the default one.

4 Decapsulation of Overlay Network Communication

Available network technologies provide ways to encapsulate various network protocols inside carrier traffic. This approach practically establishes an overlay network on top of an existing network infrastructure. The virtual topology of such an overlay network is usually different than the physical topology. Encapsulation methods can aim to maintain security *Confidentiality, Integrity, and Availability* (*CIA*) triad. As already explained, the goal of OUR TOOL is to offer an extensive forensic analysis of captured traffic. To fulfill this goal and provide a broader range of use-cases, our research and development further focused on the processing of encapsulated traffic. This section, therefore, outlines several encountered challenges and explains how the analysis of encapsulated satellite traffic was solved.

4.1 Generic Stream Encapsulation

Network protocol *Generic Stream Encapsulation* (*GSE*) was defined by the *Digital Video Broadcasting Project* (*DVB*) and it offers a way to transport IP traffic over generic physical layer, usually over DVB physical infrastructure [8, p. 6]. GSE, as a native IP encapsulation protocol on DVB bearers, was introduced with the second-generation satellite transmission system called DVB-S2 (Figure 5). Generic data transmission on the first generation of DVB standards was formerly possible using the *Multi-Protocol Encapsulation* (*MPE*) on MPEG-TS packets. However, MPE suffered significant overhead. GSE is also included in Satlabs System Recommendations for DVB-RCS terminals [23].

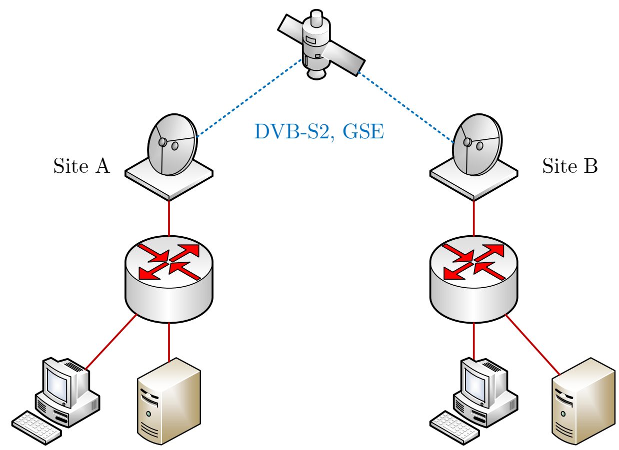

Figure 5. This example scenario is presenting a professional application of DVB-S2 and GSE. This architecture offers point-to-point or point-to-multipoint connections over a satellite link in both directions. Traffic between Site A and Site B is carried using Generic Stream Encapsulation. The figure is based on the GSE implementation guidelines [6, p. 11].

Outline of GSE Procedures Operation of GSE allows transmission of variable size generic data encapsulated into baseband frames. GSE can encapsulate not only IPv4 traffic, but a wide range of other protocols including IPv6, Ethernet, ATM, MPEG, and others. It supports addressing using 6-Byte MAC addresses, 3-Byte addresses, and even a MAC address-less mode [8, p. 6]. Encapsulation and decapsulation procedures performed by the DVB broadcast bearers are transparent to the rest of the network topology and the carried traffic. Shall a network layer PDU be transmitted over a satellite connection, GSE packets serve as a data link layer (Figure 5). This GSE layer provides encapsulation, fragmentation, and slicing. Created GSE packets are then carried in baseband frames (e.g. DVB-S2) on the physical layer (Figure 6). The receiving side performs a reassembly process, integrity check, and a final decapsulation of transmitted PDUs [4].

Moreover, it is also possible to transport GSE packets over, for example, standard IP network infrastructure. In this case, the DVB-S2 traffic can be carried like a generic payload on the application layer with the use of *User Datagram Protocol* (*UDP*) as a transport layer. Therefore, given UDP datagrams carry DVB-S2 baseband frames, which further carry

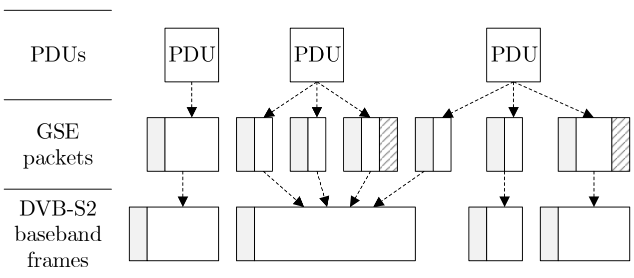

Figure 6. The figure shows the encapsulation of network layer PDUs into GSE packets and transmission of GSE packets inside physical layer baseband frames. GSE packets and baseband frames consist of a header (shown as a grey block) and a data field (shown as white space). GSE packet carrying the last fragment also contains CRC-32 (shown as a block with pattern). The figure is based on GSE protocol specification [8, p. 10].

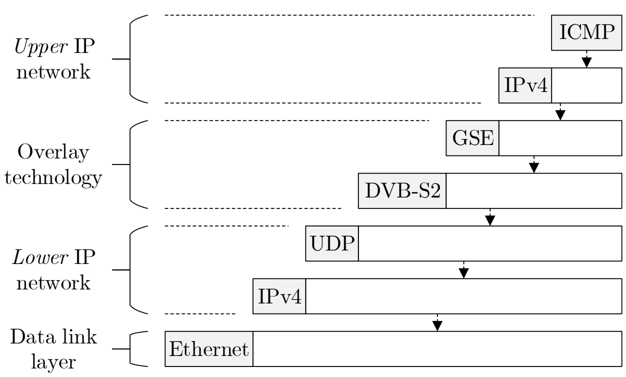

Figure 7. Example of IP traffic encapsulated in GSE layer, which is carried by another IP traffic. The resulting virtual topology can be characterized as an established overlay network.

GSE packets encapsulating selected protocol communication. This approach effectively establishes an overlay network infrastructure, because IP traffic can practically carry GSE packets, which can carry another layer of IP traffic. At this point, the UDP/IP layer *below* GSE can be considered the *carrier (encapsulating) traffic* while, for example, the IP layer *above* GSE can be described as the *carried (encapsulated) traffic*. This approach is presented in Figure 7.

According to specifications and recommendations published by SatLabs, implementation of a receiver with Ethernet interface can be divided into demodulation/decoding device, and a device focused on baseband processing. In such case, *L3 Mode Adaptation Receiver Header* can be prepended to received data [22, p. 10]. The receiving device would then process *DVB-S2 L3 Mode Adaptation Receiver Header*, *DVB-S2 baseband frame*, and *GSE packets* to analyze transmitted communication.

Fragmentation, Slicing, Padding and Reassembly Process As noted earlier, GSE procedures can encapsulate different protocol data units in one or more GSE packets. In general, GSE packets have variable length, and they can be sent in different baseband frames individually or in a group. Therefore, fragmentation, slicing, padding and reassembling can occur. In this context, fragmentation refers to a situation when a PDU and Extension Header is fragmented into multiple GSE packets (Figure 6). Slicing indicates a case when a GSE packet itself is divided into several contiguous baseband frames [8, p. 8]. Noted slicing, therefore, refers to physical layer fragmentation, which shall be transparent to the GSE layer [6, p. 27]. Concerning DVB-S2 applications, GSE slicing (fragmentation into baseband frames) does not occur [6, p. 31].

Shall a single PDU be fragmented into several GSE packets, each packet is assigned a *Fragmentation Identifier* (*Frag ID*) label in the GSE header [8, p. 17]. Frag ID is used to match fragments belonging to the same original PDU. This approach enables the simultaneous transmission of fragments from up to 256 different original PDUs. GSE packets carrying a complete PDU and GSE packets with PDU fragments can be distinguished using start and end flags in the GSE header. The protocol of carried PDU is indicated by Protocol Type/Extension field in the GSE header of the first fragmented packet and every not fragmented packet. The packet with the last PDU fragment further carries a CRC-32 field used to check integrity after the reassembly process (Figure 6). It is important to note that for example DVB-S2 allows multiplexed transmission of multiple streams, each identified by its *Input Stream Identifier* (*ISI*) [6, p. 32] in baseband header [7, p. 20]. The reassembly process has to be carried out independently for each received stream [8, p. 21]. Some of the possible GSE packet formats are presented in the technical specification [8, pp. 31–32].

Concerning GSE addressing modes noted earlier, an additional fourth mode called *label re-use* can be used when multiple GSE packets are carried in a single baseband frame. Shall label re-use be indicated, current GSE packet without address belongs to the same address as the last previously processed GSE packet. More detailed analysis of GSE protocol is beyond this paper's scope. GSE packet format is defined in the protocol specification [8, p. 12]. Further information can be found in standards, recommendations, and guidelines covering GSE and DVB-S2, [8], [9], [10], [6], [11].

Implementation Outline Our main goal was to successfully decapsulate and process GSE protocol used as an overlay network technology (Figure 7). Main challenges were represented by correct decapsulation of fragmented traffic including timeout detection and also including support for recursive encapsulation. As outlined earlier, this approach represents the transmission of following protocols layered on top of each other:

- *upper* IP as an overlay network layer,
- GSE packets transmitted inside a DVB-S2 baseband frame with Mode Adaptation Header,
- *lower* IP and UDP as a network and a transport layer,
- Ethernet as a data link layer.

Designed extension of object model concerning the processing of encapsulated communication (Figure 8) is quite straightforward and reflects above-described protocol layers. Instance of *BaseBandFrame* composes of *ModeAdaptationHeaderL3*, *BaseBandHeader*, and several user packets. These user packets are, in this case, GSE packets. Instance of *GsePacket* includes *GseHeader* and carries the encapsulated PDU. Properties of these instances store values of specific protocol fields from the processed frame, e.g., address label, length, fragment ID, encapsulated protocol type, checksum, etc. All designed model classes make use of factory methods for parsing corresponding instances from network traffic. These *Parse* methods, therefore, take an instance of *PDUStreamReader*, which is responsible for providing a correct sequence of bytes belonging to the *lower* PDU, as described above.

Because GSE packets can represent fragments of the encapsulated PDU, *GsePacket* class implements *IFragment* interface utilized during reassembly procedures. With the challenge of correct reassembly and decapsulation, a new type of network traffic frame was introduced. Class *PmFrameEncapsulated* inheriting from *PmFrameBase* represents a frame encapsulated in one or more carrier datagrams. Carrier datagrams can be either baseband frames or encapsulation packets. The instance of *PmFrameEncapsulated* has references to individual fragments which form the given frame.

Processing of GSE-encapsulated communication is managed by *L7DvbS2GseDecapsulatorBlock* (Figure 9) dynamically connected to the frame processing pipeline, which was described in Figure 4. This TPL block aims to decapsulate frames from GSE packets used as an overlay network technology. Connection to the pipeline is established using *BroadcastBlock*, which is capable of forwarding *L7Conversations* from the *L7ConversationTrackerBlock* to the *StoreL7ConversationBlock* (as in the standard pipeline topology presented in Figure 4) and also to the noted *L7DvbS2GseDecapsulatorBlock* (Figure 9). Due to the possible amount of false positive detections of GSE layer, decapsulation procedures are optional. Main OUR TOOL application settings include such option to enable *Decapsulation during capture file import* for communication of *Generic Stream Encapsulation (GSE) inside DVB-S2 baseband frames with Mode Adaptation Header L3 sent as Layer 7 PDU*. Shall this option be enabled, ControllerCaptureProcessor instantiates and connects *L7DvbS2GseDecapsulatorBlock* to the pipeline.

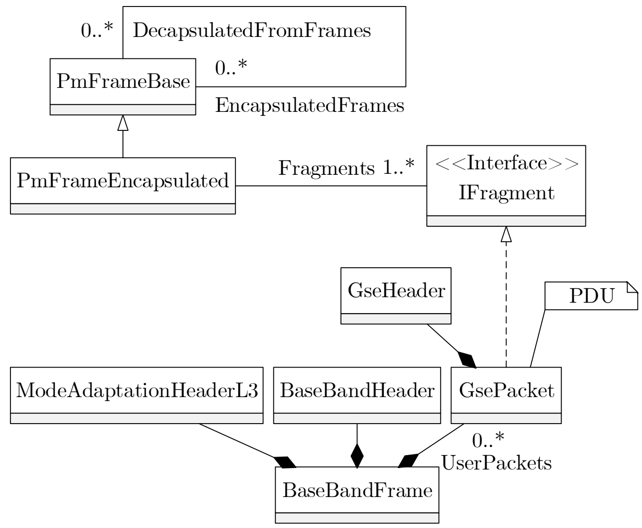

Figure 8. Extension of object model focused on the processing of GSE-encapsulated frames (simplified).

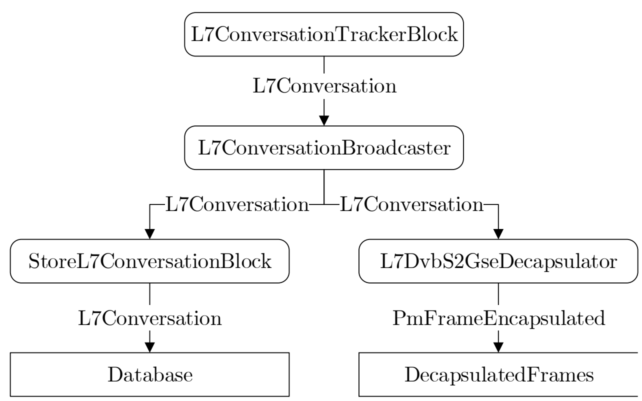

Figure 9. Scheme illustrating the connection of *L7DvbS2GseDecapsulatorBlock* to the frame processing pipeline using *BroadcastBlock* placed between *L7ConversationTrackerBlock* and *StoreL7ConversationBlock*. Standard pipeline topology is shown in Figure 4.

Because GSE packets, which can encapsulate IP traffic, can be transmitted inside another UDP/IP, recursive encapsulation can happen. In such an edge case, several GSE overlay networks could be created on top of each other. That implies that a frame decapsulated from GSE packets must be separately processed and analyzed for the presence of another GSE layer. The challenge of recursive encapsulation is handled by *ControllerCaptureProcessor*, as well. Shall the frame processing pipeline finish with some decapsulated frames, another pipeline is established, and these decapsulated frames are further processed.

The decapsulation procedure performed by *L7DvbS2GseDecapsulatorBlock* is following. Instantiated *PDUStreamReader* handles reading bytes of the input conversation and then parsing of a GSE layer is attempted. Upon successful detection of GSE layer, DVB-S2 baseband frames are passed to the *GseReassemblingDecapsulator*. It outputs frames which have type *PmFrameEncapsulated* and are ready for further processing.

The *GseReassemblingDecapsulator* manages decapsulation of frames encapsulated inside GSE packets, which are carried in baseband frames. The decapsulator is capable of reassembly procedure according to the specification [8, p. 21]. Reassembling distinguishes single input stream and multiple input streams based on *ISI* explained earlier. The reassembly procedure utilizes *GseReassemblyBuffer* for each fragment ID and for each stream identifier processed. The decapsulator, therefore, decapsulates frames from GSE packets in baseband frames. In the case of GSE fragmentation, given GSE packet (fragment) is added to the corresponding reassembly buffer. Upon successful reassembly, the carried frame is then decapsulated, too. Each *GseReassemblyBuffer* holds a counter of processed baseband frames, which is used to detect a PDU reassembly time-out error, as defined in the specification [8, p. 22].

4.2 Evaluation

Every layer of decapsulated traffic is subject to further network forensic analysis performed by the OUR TOOL. GUI with frame content informs the user whether the current frame in encapsulated or not. It is also possible to navigate the frame content view between individual carrier (encapsulating) and encapsulated frames as shown in Figure 10 and Figure 11.

The implementation has been evaluated on publicly available data [2] and results were compared to the reference Wireshark implementation. The OUR TOOL reconstructed the communication as we demonstrate on Fig. 10 and 11..

Results are supported by a set of integration tests that verify correct processing of GSE traffic in future releases and prohibits regression bugs to be introduced [LINK REMOVED DUE TO DOUBLE-BLIND REVIEW].

4.3 Limitations

Our main goal was to process GSE traffic used as an overlay network technology. Therefore, the current implementation of GSE decapsulation does not support processing of DVB-S2 baseband frames directly as a physical layer. The decapsulation procedure also does not take GSE labels into account, because OUR FRAMEWORK does not support tracking multiple L1 conversations. Stream ID and fragment ID is correctly utilized during GSE reassembling. However, neither stream ID is used to separate L1 conversations. To conclude,

[2] https://wiki.wireshark.org/DVB-S2 (last accessed 2019-04-17).

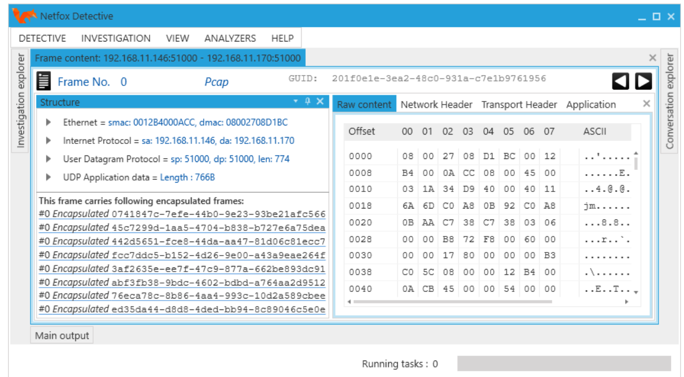

Figure 10. View of the frame content of the OUR TOOL presenting a frame carrying eight other encapsulated frames. It is possible to navigate between encapsulated frames using shown links labeled with GUID of the target frame.

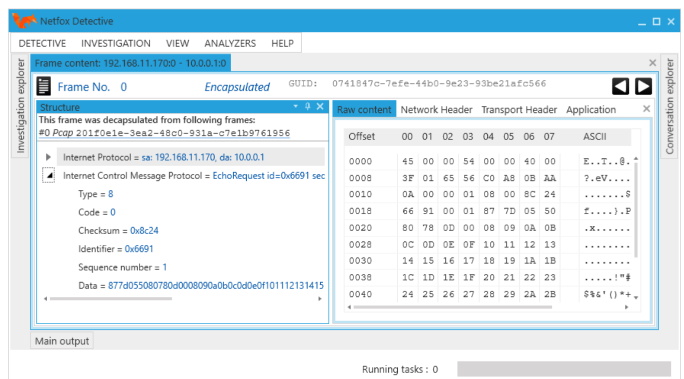

Figure 11. Frame content view of OUR TOOL (as in Figure 10) analyzing a frame that was decapsulated from another frame of *lower* layer.

we focused on processing IP traffic encapsulated in GSE inside DVB-S2 baseband frames with Mode Adaptation Header L3 sent as Layer 7 PDU of UDP/IP.

5 Conclusion

We have implemented proof-of-concept support of GSE for OUR TOOL. All source codes are open-source and publicly available on [Link removed due to double-blind review]. The OUR TOOL becomes the first of NFATs with support of GSE. Our implementation was evaluated against the only known implementation in NSM tool — Wireshark. This way, we enriched OUR TOOL with carving capabilities by the support of a new data-source and demonstrated its extensibility to support new protocols on all network layers.

OUR TOOL is publicly available [LINK REMOVED DUE TO DOUBLE-BLIND REVIEW] for all network forensic practitioners to use, including open-source source codes to be modified, or be incorporated into other tools the investigators use.

References

[1] Nicole Beebe. 2009. Digital forensic research: The good, the bad and the unaddressed. In *IFIP International Conference on Digital Forensics*. Springer, 17–36.

[2] Eoghan Casey. 2004. Network traffic as a source of evidence: tool strengths, weaknesses, and future needs. *Digital Investigation* 1, 1 (2004), 28 – 43. https://doi.org/10.1016/j.diin.2003.12.002

[3] M. I. Cohen. 2008. PyFlag - An Advanced Network Forensic Framework. *Digital Investigation* 5 (Sept. 2008), 112–120. https://doi.org/10.1016/j.diin.2008.05.016

[4] DVB Project Office. 2015. *DVB-GSE - Generic Stream Encapsulation*. DVB Project Office. URL: https://www.dvb.org/resources/public/factsheets/dvb-gse_factsheet.pdf.

[5] Sandstorm Enterprises. 2003. NetIntercept.

[6] ETSI. 2011. *ETSI TS 102 771 V1.2.1 - Digital Video Broadcasting (DVB); Generic Stream Encapsulation (GSE) implementation guidelines*. European Telecommunications Standards Institute. URL: https://www.etsi.org/deliver/etsi_ts/102700_102799/102771/01.02.01_60/ts_102771v010201p.pdf.

[7] ETSI. 2014. *ETSI EN 302 307-1 V1.4.1 - Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2*. European Telecommunications Standards Institute. URL: http://www.etsi.org/deliver/etsi_en/302300_302399/30230701/01.04.01_60/en_30230701v010401p.pdf.

[8] ETSI. 2014. *ETSI TS 102 606-1 V1.2.1 - Digital Video Broadcasting (DVB); Generic Stream Encapsulation (GSE); Part 1: Protocol*. European Telecommunications Standards Institute. URL: https://www.etsi.org/deliver/etsi_ts/102600_102699/10260601/01.02.01_60/ts_10260601v010201p.pdf.

[9] ETSI. 2014. *ETSI TS 102 606-2 V1.1.1 - Digital Video Broadcasting (DVB); Generic Stream Encapsulation (GSE); Part 2: Logical Link Control (LLC)*. European Telecommunications Standards Institute. URL: https://www.etsi.org/deliver/etsi_ts/102600_102699/10260602/01.01.01_60/ts_10260602v010101p.pdf.

[10] ETSI. 2014. *ETSI TS 102 606-3 V1.1.1 - Digital Video Broadcasting (DVB); Generic Stream Encapsulation (GSE); Part 3: Robust Header Compression (ROHC) for IP*. European Telecommunications Standards Institute. URL: https://www.etsi.org/deliver/etsi_ts/102600_102699/10260603/01.01.01_60/ts_10260603v010101p.pdf.

[11] ETSI. 2015. *ETSI TR 102 376-1 V1.2.1 - Digital Video Broadcasting (DVB); Implementation guidelines for the second generation system for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2; Part 1: DVB-S2*. European Telecommunications Standards Institute. URL: https://www.etsi.org/deliver/etsi_tr/102300_102399/10237601/01.02.01_60/tr_10237601v010201p.pdf.

[12] Dan Farmer and Wietse Venema. 2009. *Forensic Discovery* (1st ed.). Addison-Wesley Professional.

[13] Simson L Garfinkel. 2010. Digital forensics research: The next 10 years. *Digital Investigation* 7 (2010), S64–S73.

[14] Vikram S Harichandran, Frank Breitinger, Ibrahim Baggili, and Andrew Marrington. 2016. A cyber forensics needs analysis survey: Revisiting the domain's needs a decade later. *Computers & Security* 57 (2016), 1–13.

[15] P. Leach, M. Mealling, and R. Salz. 2005. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. http://www.ietf.org/rfc/rfc4122.txt

[16] Petr Matoušek, Jan Pluskal, Ondřej Ryšavý, Vladimír Veselý, Martin Kmeť, Filip Karpíšek, and Martin Vymlátil. 2015. Advanced Techniques for Reconstruction of Incomplete Network Data. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* 2015, 157 (2015), 69–84. http://www.fit.vutbr.cz/research/view_pub.php?id=10864

[17] Moloch. cited April 2019. URL: https://molo.ch.

[18] NetworkMiner. cited April 2019. URL: https://www.netresec.com/?page=NetworkMiner.

[19] NfDump. cited April 2019. URL: https://github.com/phaag/nfdump.

[20] Emmanuel S. Pilli, R. C. Joshi, and Rajdeep Niyogi. 2010. Network Forensic Frameworks: Survey and Research Challenges. *Digital Investigation* 7, 1-2 (Oct. 2010), 14–27. https://doi.org/10.1016/j.diin.2010.02.003

[21] PyFlag. cited April 2019. URL: https://github.com/py4n6/pyflag.

[22] SatLabs Group. 2008. *Mode Adaptation Input and Output Interfaces for DVB-S2 equipment Version 1.3*. SatLabs Group. URL: http://satlabs.org/pdf/sl_561_Mode_Adaptation_Input_and_Output_Interfaces_for_DVB-S2_Equipment_v1.3.pdf.

[23] SatLabs Group. 2010. *SatLabs System Recommendations Version 2.1.2*. SatLabs Group. URL: http://satlabs.org/pdf/SatLabs_System_Recommendations_v2.1.2_final.pdf.

[24] Suricata. cited April 2019. URL: https://suricata-ids.org.

[25] TCPDUMP. cited April 2019. URL: https://www.tcpdump.org/.

[26] Tcpdump/Libpcap. 2018. LINK-LAYER HEADER TYPES. https://www.tcpdump.org/linktypes.html

[27] TCPFlow. cited April 2019. URL: https://github.com/simsong/tcpflow.

[28] Wireshark. cited April 2019. URL: https://www.wireshark.org/.

[29] XPlico. cited April 2019. URL: https://www.xplico.org/.

[30] Zeek. cited April 2019. URL: https://www.zeek.org.