*electronics*

MDPI

*Article*

# LAN Traffic Capture Applications Using the Libtins Library

**Adrian-Tiberiu Costin, Daniel Zinca * and Virgil Dobrota**

Communications Department, Technical University of Cluj-Napoca, 400114 Cluj-Napoca, Romania; costin.t.adrian@gmail.com (A.-T.C.); Virgil.Dobrota@com.utcluj.ro (V.D.)
* Correspondence: daniel.zinca@com.utcluj.ro

**Abstract:** Capturing traffic and processing its contents is a valuable skill that when put in the right hands makes diagnosing and troubleshooting network issues an approachable task. Apart from aiding in fixing common problems, packet capture can also be used for any application that requires getting a deeper understanding of how things work under the hood. Many tools have been developed in order to allow the user to study the flow of data inside of a network. This paper focuses on documenting the process of creating such tools and showcasing their use in different contexts. This is achieved by leveraging the power of the C++ programming language and of the libtins library in order to create custom extensible sniffing tools, which are then used in VoIP (Voice over IP) and IDS (Intrusion Detection System) applications.

**Keywords:** Apache Kafka; IDS sensors; ksqlDB; libtins; RTP; SIP; VoIP



**Citation:** Costin, A.-T.; Zinca, D.; Dobrota, V. LAN Traffic Capture Applications Using the Libtins Library. *Electronics* **2021**, *10*, 3084. https://doi.org/10.3390/electronics10243084

Academic Editor: Juan-Carlos Cano

Received: 9 November 2021
Accepted: 3 December 2021
Published: 11 December 2021



## 1. Introduction

Capturing the data that flow throughout a network is extremely important because it allows the user to intercept, view and analyze network packets. This grants us the ability to get a better grasp of what happens under the hood, from viewing the protocol stack that is used to seeing the individual bytes of data that are being sent and received. The tools that allow us to perform this analysis are labeled as traffic sniffers, packet capture applications or protocol analyzers.

A sniffer, or protocol analyzer, has many uses, from monitoring bandwidth and traffic patterns and exploring the flow of conversations throughout the network to troubleshooting and solving problems as they occur. Furthermore, from a security standpoint, packet sniffing is often used to intercept conversations between users and view the data with the purpose of scanning it for malicious activity or to detect holes in the networks' security. If the person capturing the traffic has ill intentions, the data can also be used to eavesdrop on the conversations between users.

Packet capture tools can be written using a variety of different programming languages. This research paper focuses on showcasing how such a tool can be built using the C++ programming language and the libtins library.

The research extends in two different directions. Firstly, the conversation is steered toward extending the libtins [1,2] packet sniffing library by adding support for parsing and processing protocols such as SIP (Session Initiation Protocol), SDP (Session Description Protocol) and RTP (Real-time Transport Protocol). These protocols are used in IP (Internet Protocol) telephony in order to produce an application that intercepts VoIP calls and outputs signaling logs and audio information. The second part of the paper is focused on producing a custom sniffer that outputs network packets in JSON (JavaScript Object Notation) format, which are used as input to an IDS in order to detect different types of attacks.

A different approach (that uses mirrored traffic from OpenFlow switches) to the same set of problems is described in [3] and focuses on the loss problems caused by the aggregation of mirrored flows from switches. Our implementation uses the raw traffic

captured by the network adapters in the computers running our software instead of extracting information from the OpenFlow protocol.

### *1.1. Libtins*

"Libtins is a high-level, multiplatform C++ network packet sniffing and crafting library" [1,2]. The objects used for capturing network packets are defined using classes such as Sniffer and FileSniffer. The first is for intercepting packets from the network interface, whilst the second one is for reading packets from PCAP (Packet Capture) files. These datagrams are then stored as PDU (Protocol Data Unit) objects, allowing further processing in a very short amount of time. This is similar to libpcap [4], a more common library, but it offers packet analysis capabilities that might be more suitable than it. According to [5,6], libpcap is considered a reference library in the field of network traffic capture and analysis. However, in [7,8], libtins was analyzed compared to other solutions. It seems that libtins is more suitable than libpcap to capture WLAN traffic [9] and to generate a larger number of IPv6 packets [10] or MQTT packets in an IoT network for security testing [11]. Based on our own experience, initially communicated in [12], we wanted to have a deeper evaluation of libtins and its performance and to demonstrate its better capabilities in some applications compared to the competing solutions.

#### 1.1.1. PDU Class

PDU is an abstract class, which acts as a base from which all other protocols are implemented inside the library. Each new protocol inherits its main attributes directly from the PDU class, such as default parameters and methods that return basic information such as length and type of PDU, for example.

Some extensions of the PDU class are provided inside the Tins namespace in order to define commonly used protocols for different networking layers, with a few examples including DNS (Domain Name System), ICMP (Internet Control Message Protocol), IP, UDP (User Datagram Protocol), DHCP (Dynamic Host Configuration Protocol) and many more. The Tins namespace contains everything the library has to offer, from the definition and implementation of each protocol to the packet sniffing and crafting capabilities using constructs such as Sniffer, FileSniffer, PacketWriter, PacketSender and so on. Even though pre-built functionality already exists with every build of libtins, extending the library is encouraged. This can be done by creating user defined PDUs, which is discussed in the Section 2 of the paper, where we describe the process of creating custom classes for SIP and RTP network packets.

#### 1.1.2. PacketSender and PacketWriter Class

The library allows the user to store the generated PDU objects in output files by using the PacketWriter class. It also allows sending the packets back on the network by using the PacketSender class. This grants the user the ability to store the created/captured packets or manipulate them and further use them for communication on the network.

#### 1.1.3. Sniffer and FileSniffer Class

The Sniffer and FileSniffer classes grant the libtins library packet capture capabilities. Objects of this type allow sniffing either from a live network interface or from "pcap" files and can be configured with libpcap filters. The captured packets are interpreted and stored in PDU objects.

In order to use the Sniffer or FileSniffer class, the user first needs to set the capture interface or specify an input file. Optionally, they can also specify a capture filter to narrow down on the captured packets. After this has been done, the incoming packets can be intercepted in one of two ways, either one at a time, using the next packet method, or by using the sniff loop method that keeps capturing packets until it is stopped manually, or an error is encountered.

#### 1.1.4. Processing Captured Packets

Every time a packet is captured, a PDU object is created from it. This object contains all the data from the network packet, and the user can use specific methods such as find_pdu to search for information specific to a certain layer. For example, if a DNS packet is captured, the created PDU would be made up of EthernetII/IP/UDP/RawPDU. As one can observe, each PDU has an inner PDU that contains the upper layer protocol, up to the RawPDU which holds the payload data. If the user wishes to process the payload, special functions such as raw_to <PDU_Type> can be called to use the raw payload data as input for the construction of other protocol objects.

Many protocol types, including some application layer protocols, are handled by default by the libtins library, but nothing prevents the user from experimenting with the above method and building his/her own application layer protocol objects. Creating custom application layer protocols that are not defined in the library is further detailed in the Section 2 of this paper, as we used this method during our research to extend the library to capture and process SIP and RTP packets in order to intercept VoIP calls.

### *1.2. Extending the Libtins Library with SIP and RTP Classes for VoIP Packet Analysis*

Using technologies such as VoIP and others to start different types of communication sessions (voice calls, video calls, meetings, etc.) is now easier than it has ever been. However, like any piece of technology, communication over packet-switched IP networks comes with certain drawbacks in the form of latency, malformed packets, reception of packets that are not in the same order as they were sent in, jitter and more. Furthermore, if the network security is not configured properly, the sessions can be intercepted by a third party [13,14].

These drawbacks introduce technical problems that require network and service management solutions in order to be fixed. One of the aims of this research was to explore ways of approaching the problems previously listed. A solution would be to use a packet sniffing program such as Wireshark [15,16] for analyzing and identifying network issues such as in [17,18] or its command-line equivalent, tshark, such as in [19,20]. This, however, is sometimes not possible due to the limitations of the target device on which the application is being run or simply because the user wishes to use a less resource-intensive alternative. In such a case, a command-line application, aimed at processing only VoIP packets would be better suited for the job because it is more lightweight. Additionally, a custom packet capture application allows for further processing of traffic, for example, by adding metadata to identify a certain flow. The ability to easily modify the solution we developed is a major advantage because, in the future, it will allow us to achieve more than just packet capture, for which a tool such as Wireshark or tshark would have been sufficient. Further goals are discussed in more detail in Section 1.3.4.

Researching how to develop a packet capture application tailored to processing the audio and signaling data of VoIP calls led us to find the libtins packet capture library, which we used as a base on top of which the extra functionality for parsing application layer protocols such as SIP and RTP was built.

Before discussing the implementation details that need to be taken into consideration when extending the libtins library with functionality for VoIP application layer protocols, we must first discuss how the IP telephony network works.

Similar to other telecommunication networks, IP-based voice and multimedia communications rely on a number of systems to be in place so that a proper data session can be established and maintained. This paper is focused on the signaling system used to establish a valid session between IP network users and the transport protocol involved in the data exchange between the users.

The gist of how a VoIP network works is that the users, also called the clients, need IP phones to connect to the server. In order to establish the session, in our approach, they exchange signaling messages: Session Initiation Protocol (SIP) for call control and Session Description Protocol (SDP) for bearer control. After the session has been established, the

Real-time Transport Protocol (RTP) is used for transmitting the multimedia data from user to user.

#### 1.2.1. Session Initiation Protocol

The Session Initiation Protocol is used inside an IP telephony network very much in the same way the SS7 (Signaling System 7) was used in the circuit-switching-based telecommunications networks: Integrated Services Digital Network (ISDN), Global System for Mobile Communications (GSM), etc. The protocol facilitates establishing and initiating multimedia data sessions by exchanging signaling messages, as it was defined in RFC3261 [21]. The commands are in ASCII format and were inspired by Hypertext Transfer Protocol (HTTP) [21]. The structure consists of one request or response line, depending on the type of the packets followed by multiple header fields, some of which are mandatory and some optional, and a message body containing SDP information if present.

By default, SIP runs over UDP at port 5060 and in call establishment, INVITE messages are sent. Based on that, paper [22] introduced a tool developed in C based on a generic packet sniffer [23] using raw BSD sockets in Linux. It captures SIP packets by selecting UDP segments and parsing the payload for 5060 and either 1 INVITE or 2 INVITE. Compared to [22], our solution extracts all SIP packets and decodes the entire SIP header structure.

Because by default the messages are not authenticated or encrypted, one important area that must be covered is security [24]. Our tool captures SIP traffic that is not encrypted.

#### 1.2.2. Session Description Protocol

Conveying media details, transport addresses and other session description metadata to the participants is done using the Session Description Protocol. This provides a standard representation for the information mentioned previously, irrespective of how the multimedia information is being transported. The protocol was defined in RFC4566 [25], and a message has three sections related to the description of session, timing and media. It is possible to have several timing and media descriptions.

#### 1.2.3. Real-Time Transport Protocol

This protocol provides end-to-end delivery of data in real-time. This makes it suitable as the transport protocol for the interactive audio and video communication used in VoIP networks, according to RFC3550 [26]. The structure of RTP consists of the header followed by extensions if they are present and a data field containing the exchanged multimedia information.

### *1.3. Building an IDS Using Libtins, Apache Kafka and ksqlDB*

One of the most important requirements in modern computer networks is related to security. Mechanisms designed for detecting or preventing intruders are paramount to the safety and reliability of the network. When implemented properly, such a system can ensure secure and trusted communications between organizations.

An IDS is used to monitor network traffic for suspicious activity and issue alerts when it is detected. This is done by using a sensor (network probe) that gets triggered based on certain patterns and prompts the system to send an alarm to an administrator or collect the data in a centralized location for later analysis. Being a passive monitoring device, the IDS comes with the advantage of having no impact on the actual network at the cost of providing no actual active protection because it relies on another system to act in case of an attack. One can build such a system by following the instructions provided in [27].

In the following paragraphs, we introduce some of the components required for building an IDS using the following tools. The first one is called Vagrant [28], needed to set up and provision virtual machines. These were used in a particular manner, as described in [27], in order to set up a virtual network to run our simulations and tests. The second tool is called Apache Kafka, needed to ingest the captured data packets and analyze them.

As part of the Kafka ecosystem, ksqlDB is the third one, and it was used to detect intruders and malicious activity inside the virtual network.

#### 1.3.1. Vagrant

Users can benefit from this tool because it can be involved in establishing a work environment, which should be portable, reusable and easy to be configured [28]. The architecture has Provisioners and Providers as building blocks to manage the development environments.

A Provisioner allows the user to customize the configuration of virtual environments (e.g., Puppet, Chef and Ansible). On the other hand, a Provider is a service that Vagrant uses to set up and create virtual environments (e.g., Oracle VirtualBox, KVM, Hyper-V and others).

The main role of Vagrant is to sit on top of virtualization software as a wrapper and help the developer interact easily with the providers, automating the configuration of the virtual environments. These configurations are placed inside a Vagrantfile and are executed step by step in order to create a development-ready box.

#### 1.3.2. Apache Kafka

Apache Kafka simplifies the communication between multiple systems using the publisher/subscriber pattern [29]. Therefore, the Kafka Cluster represents the communications backbone that decouples the systems. A producer publishes data on one or multiple streams (or channels). A stream is unidirectional. The streams are stored in commit logs which are append-only structures that keep all capture events in an ordered sequence. Once appended in the log, a certain record will keep the same offset (as the position of the record from the beginning of the log). Appending a record makes it immutable. An update of an existing record will create another one in the log, keeping the original one unmodified [30]. Kafka is an implementation of a software bus that uses stream processing. It is an open-source software platform written in Scala and Java, which aims to provide a unified, high throughput, low-latency platform for handling real-time data feeds. It can also connect to external systems for importing and exporting data with Kafka Connect, and it needs Kafka Streams, a Java stream processing library.

Kafka was designed to store key-value messages that come from processes called producers. The data can be partitioned into different partitions with different topics. Other processes called consumers can read messages from these partitions. The five major APIs it provides are the following: (1) producer (allowing an application to publish a stream of records); (2) consumer (an application can subscribe to topics and processes streams of records); (3) connector (executing the reusable producer and consumer APIs that can link the topics to the existing applications); (4) stream (converting the input streams to output and producing the results); and (5) admin (to manage Kafka topics, brokers and other objects). The whole architecture runs on a cluster of one or more so-called brokers, and the partitions are distributed across the cluster nodes and are replicated to multiple brokers. This architecture allows Kafka to deliver massive streams of messages in a fault-tolerant fashion. Because of this, it has successfully replaced some of the conventional messaging systems such as Java Messaging Service (JMS), Advanced Message Queueing Protocol (AMQP) and others. In addition to generic streaming applications [31–34], Kafka can also be used in Intrusion Detection Systems [35].

#### 1.3.3. ksqlDB

This is a type of database for stream processing applications [36]. It achieves this by consolidating the many components that are found in almost every stream processing architecture.

Almost all streaming architectures today require a subsystem to acquire events from existing data sources, another subsystem to store those events, another one to process them and another one to serve queries against aggregated materializations. Integrating these subsystems can be difficult and because of the many mental models, a lot of complexity is

introduced. For ksqlDB there is just one mental model for doing everything that is needed. Thus, the whole application depends on Apache Kafka only [36].

Because of ksqlDB's ability to extract information in real time from a running flow, it can be used for a variety of applications, from energy consumption monitoring [37] to processing industrial sensor data [38].

#### 1.3.4. Putting It All Together

In the implementation part of the paper, we discuss how to combine the technologies described above, along with the libtins library and the C++ programming language to create an IDS. The strategy follows the original confluent.io article [27], but the major novelty is that we replaced the tool they chose for packet capture (tshark) with a custom sniffer that intercepts network traffic, creates and stores JSON objects and feeds them into Apache Kafka. The goal we had in mind here was to create a small, lightweight, extensible alternative to tshark, aiming to further develop it in the future to fit our use case. The main advantage of having this type of tool as opposed to simply using a readily available protocol analyzer is that we have the possibility of easily modifying the source code to add extra functionality based on our needs. Our next goal will be to extend this custom tool by computing and adding metadata that uniquely identifies the flow, before inserting it into the Kafka database. This will allow for faster processing of IDS rules directly from the database. Furthermore, we are researching ways of directly sending the captured network data from our application to Apache Kafka. Thus we do not need to rely on intermediate storage such as the current tshark-based solution, which dumps all the sniffer output into a shared directory, from where it is later fed into Apache Kafka using a connector.

According to [39–41], the VoIP traffic in 4G/5G networks requires high data rates and is still vulnerable to attacks. The tools and methodologies we present in this paper can be used as described to capture VoIP traffic in 5G and B5G (Beyond 5G) networks.

## 2. Implementation

This section describes the implementation details of extending the libtins library for different applications. The first part is focused on the development of the VoIP analysis command-line interface, which extracts information from SIP/SDP/RTP packets. The latter part discusses the software developed for capturing network traffic and sending it to Apache Kafka for intrusion detection purposes.

### *2.1. Extending the Libtins Library with SIP and RTP Classes for VoIP Packet Analysis*

Adding functionality for working with SIP and RTP packets was done by defining the structure of the previously enumerated protocols inside separate classes. Afterward, the newly defined objects were created using the data captured by the libtins library. Intercepting the data was done inside a loop, which was created by using a Sniffer object to call the sniff loop function. Each captured packet was processed inside a user-defined callback function.

Knowing that SIP and RTP are Application Layer protocols, the callback function was used to fetch the packet payload data and parse it inside the individual class constructor for each of the two protocols. After the processing was done, the packets were stored inside appropriate data structures for later decoding, in the case of RTP packets, or for easy writing to output text files, in the case of SIP packets.

All the captured packets were also written to temporary PCAP files. Taking this approach, we separated the packet capture from the packet processing, which is the recommended approach for performance reasons. Storing the packets inside a packet capture file or appropriate data structure and processing them separately either on different threads or sequentially was less error-prone and faster.

### 2.1.1. Application Diagram

The entry point for the application is the start function, which is called from main. This redirects the user to either packet creation or packet capture through a text-based user interface.

The packet creation was done by obtaining the input from the user and concatenating it into a string and using it as an argument for the SIP constructor. If the constructed packet was valid, it was sent over the network by the send_packet method inside the PacketCrafter class. The packet capture was done by creating a sniffing environment using the libtins library on either a network interface or an input file. After the packets were captured, they were processed, and output files were generated. SIP processing was straightforward while the RTP stream was converted to linear PCM through the decode function and written to an audio file.

Figure 1 shows that the application splits into two different execution paths. Path A was selected if the user entered the capture mode, listening for incoming SIP and RTP packets, processing them and producing output in the form of text and audio files. Path B was selected if the user chose to create SIP packets, verify, and send them on the network.

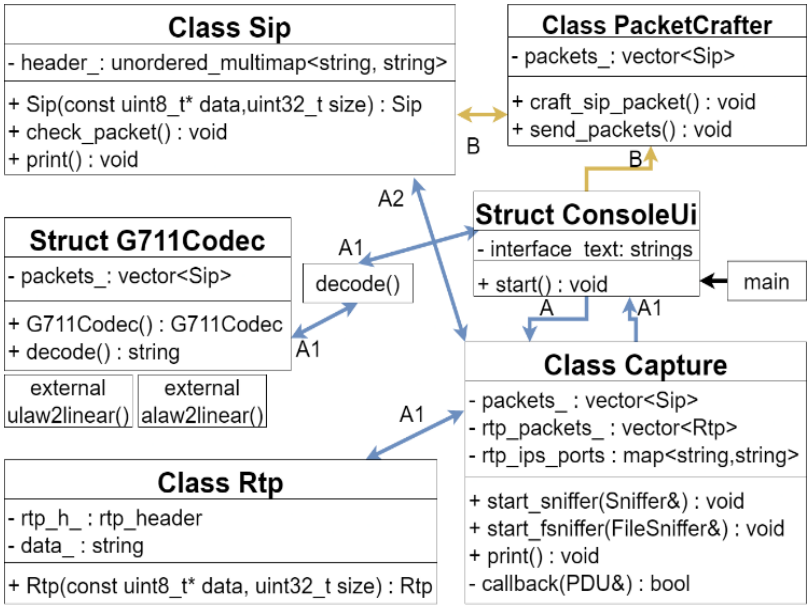


**Figure 1.** Simplified software diagram illustrating the structure of the VoIP sniffer application.

The following sections go deeper into explaining the role of each software component and how they collectively achieved the ability to capture VoIP calls. The source code for the application can be found in the GitHub repository [42].

### 2.1.2. ConsoleUI Structure

This structure was used to statically define all the messages the user sees in the text user interface during the execution of the program. These messages were arranged into two groups, one for the packet capture mode and one for the packet crafting mode.

Besides the text information, the ConsoleUI structure also held the start function, which was the entry point for our command-line application and contained all the high-level logic for its execution and user interaction logic. This function also provided the user with the option to choose between the packet capture mode and the packet crafting mode.

If the user chose to enter the packet capture mode, three options were offered, consisting of either live capture or parsing pcap files. Figure 2 shows the process of reading SIP packets from a capture file (a slightly modified version was used in a similar fashion to process RTP packets, with the exception that those packets had to run through a decoding algorithm, which is discussed in a later section, to obtain the audio information). After

the initial pass, where all live-captured network packets were processed and stored in a temporary capture file, this technique was used to read and process those packets and build SIP and RTP objects from them.

```
if(user_input == "output sip")
{
    //configure file sniffer to capture SIP packets
    config.set_filter("port 5060");

    Tins::FileSniffer sip_fsniffer(
                          "../../temp/all_traffic.pcap", config
                          );

    Tins::FileSniffer sip_fsniffer(
                          "../temp/all_traffic.pcap", config
                          );

    //parse the file for SIP packets and write the
    //output to a file
    std::cout<<"Generating SIP output files...\n";
    Capture capture_sip(Capture::IS_SIP, "sip_packets");
    capture_sip.run_file_sniffer(sip_fsniffer);

    std::string output_path = "../../outputs/sip/packet_";

    std::string output_path = "../outputs/sip/packet_";

    capture_sip.print(output_path);

    std::cout<<"SIP output files have been written to: "
             << output_path << "\n";
    break;
}
```

**Figure 2.** Reading SIP packets from capture files using the libtins library.

### 2.1.3. Capture Class

For further abstraction, a wrapper for the packet capturing functionality of the libtins library was defined in the form of the Capture class. This contained all the logic for running the Sniffer in order to capture packets. After the data had been parsed, the Capture object used std::vector objects for storage.

Using a Capture object allowed calling the run sniffer method for sniffing loop and for listening to the appropriate incoming packets. This was performed according to the capture filter, and a callback function was called for each captured packet. The header file that contained the definition for the Capture class can be seen in Figure 3.

The callback function differentiates based on the type of packets being captured and processes them accordingly. Three branches were defined inside the function, one for working with all incoming captured packets, another one only for the SIP packets and another dedicated only to RTP packet processing.

In the first case, when all the packets were captured, the program did not process them at all, just storing them by writing into an intermediary packet capture file for later use. This was done during the live network capture mode, similar to the Wireshark packet analyzer. The user could stop the live capture at any time by pressing a key on the keyboard.

In the second and third cases, when processing SIP and RTP packets, the callback function retrieved the payload of the captured packet and converted it from a RawPDU object to either a SIP or RTP instance, calling the specific constructor that constructed an object from the raw data. The processed SIP and RTP packets were stored inside data structures of type vector for later processing and were also written to temporary capture files. In addition, the source and destination IPs and UDP ports were also stored for each multimedia packet in order to be used in the RTP decoding process.

```
class Capture{
    public:
        enum CaptureType{IS_SIP=0, IS_RTP, IS_OTHER};
        bool loop_stop = false;

        Capture(CaptureType c, const std::string& filename);
        void run_sniffer(Tins::Sniffer& sniffer);
        void run_file_sniffer(Tins::FileSniffer& fsniffer);
        void print() const;
        void print(std::string& path) const;

        std::vector<Rtp> get_rtp_packets();
        std::vector<Sip> get_sip_packets();
        std::map<std::pair<std::string, std::string>,
            std::pair<std::string, std::string>> get_ports();
    private:
        std::unique_ptr<Tins::PacketWriter> p_writer;

        //sip packets
        std::vector<Sip> packets_;
        //rtp packets
        bool std::vector<Rtp> rtp_packets_;

        //map for RTP ip and port
        std::map<
            std::pair<std::string, std::string>,
            std::pair<std::string, std::string>
                > rtp_ips_and_ports;

        bool  capture_sip, capture_rtp;

        bool callback(const Tins::PDU& pdu);
};
```

**Figure 3.** Header file containing the definition of the Capture class.

#### 2.1.4. SIP Class

The SIP class defined the structure of an SIP packet and operations performed on it. Being a text-based protocol, the payload data were used to construct a large string that was later parsed line by line into the specific fields in the SIP header. Each field was stored in an unordered multimap data structure, in the form of key-value pairs where each key was the name of the specific header field and the value it points to was the header field value. The public methods and variables handled the creation, output and data retrieval that were required for working with the individual SIP packets inside the application. Besides this, we observed multiple private methods and variables that handle internal data storage and manipulation for the SIP header and the validation necessary for verifying SIP packets upon creation inside the packet crafter algorithm. The header file containing the definitions for the SIP class can be seen in Figure 4.

Because the data structure used for packet storage did not maintain the key-value pairs in the order they were inserted in, an auxiliary data structure of type vector was used to keep this order. This was needed later by the function that reconstructed the packets and generated text files for each.

Objects created from user data were built in the same way, meaning that after the user inserted all the data from the keyboard or from a text file, the input was concatenated into a string that was processed as described above. The user-created packets were then checked to validate whether they contained the mandatory header fields and correct syntax.

Creation and verification of SIP objects as described in the above paragraph were all done using the PacketCrafter class, which was implemented to craft SIP objects and acted as a wrapper for the libtins packet sending functionality. If the created SIP packet was valid, it was then sent on the network using an instance of the libtins PacketSender class, inside the send packet method.

```
class Sip
{
    public:
        //Constructor for real time captured data
        Sip(const uint8_t* data, uint32_t size);

        //Constructor for text file data
        Sip(const std::string& data);

        enum PacketType {NONE=0, REQUEST, RESPONSE}type;
        std::vector<std::string> get_header_order() const;
        std::unordered_multimap<std::string,
                        std::string> get_header() const;

        void print() const;
        void print(std::string path, unsigned p_num) const;

        //check if header has mandatory fields
        void check_packet(const std::string& filename, bool check);
    private:
        std::vector<uint8_t> buffer_;

        //key order for multimap
        std::vector<std::string> h_order_;

        std::unordered_multimap<std::string, std::string> header_;

        //check mandatory headers
        void check_headers(
                    const std::string& method_name,
                    const std::string& request_line,
                    const std::string& path);

        void check_sdp(const std::string& path);

        //pass param for header being checked to from and via
        void check_uri(std::vector<std::string>& request_uri_fields,
                const std::string& path, const std::string& method,
                const std::string& header_field);
};
```

**Figure 4.** Header file containing the definition of the SIP class.

### 2.1.5. RTP Class

The RTP class defines the structure of an RTP packet and operations performed on it. The header file that defines the structure of an RTP object can be seen in Figure 5. As we can observe, the public section of the class implementation handles RTP packet creation, along with RTP data retrieval. Inside the private implementation, we can see the internal definition of the RTP header which would be used to construct the RTP object.

```
class Rtp
{
    public:
        Rtp(const uint8_t* data, uint32_t size);
        Rtp();
        ~Rtp();

        uint8_t get_payload_type();
        uint16_t get_seq_no();
        uint32_t get_timestamp();
        std::string get_data();
    private:
        struct rtp_header
        {
            uint8_t cc : 4;
            uint8_t x  : 1;
            uint8_t p  : 1;
            uint8_t v  : 2;
            uint8_t pt : 7;
            uint8_t m  : 1;
            uint16_t seq;
            uint32_t timestamp;
            uint32_t ssrc;
            uint32_t csrc[];
        };
        rtp_header* rtp_h_p_;

        std::string  data_;
        //this will be the actual hard copy of the header
        rtp_header rtp_h_;
};
```

**Figure 5.** Header file containing the definition of the RTP class.

To construct an object of type RTP we had to first define the structure of the header. The first 12 bytes were occupied by the actual header fields, containing important information that would be used in the decoding process such as payload type, sequence number and timestamp, followed by zero or more extension headers. Knowing this, we could cast the pointer to the captured data inside the callback function to a rtp header pointer so that we could access each individual header field and store it for later use. After storing the header, we incremented the memory address pointing to the data by the size of the header and its extensions to reach the data block, which we stored inside a string that would be decoded later. The above process can be seen in Figure 6 with the full algorithm being available in the source code [39].

```
Rtp::Rtp(const uint8_t* data, uint32_t size)
{
  //store rtp header
    rtp_h_p_ = (rtp_header*) data;
  //go forward in memory by sizeof header
    data += sizeof(rtp_header);

    uint16_t extension_length = 0;

  //if we have an extension header
    if(rtp_h_p_->x)
    {
    //go forward two bytes from end of header to reach extension length
        data += 2;

  //cast the data to a unsigned short pointer and dereference it
  //ntohs() takes a unsigned short as a parameter and converts it
  //from network byte order(big endian) to host byte order(little endian)
        extension_length = ntohs(*(uint16_t*)data);

  //advance by 2 bytes(extension_length) to reach the extension header
  //and another 4 bytes(32 bits) for each increase in extension length
  //example: if the length of the extension is 1 we would advance 4 bytes
  //reach the end of the full header
        data += sizeof(extension_length) + extension_length * 4;

  //update extension length
        extension_length += extension_length * 4 + 2;
    }
    data_ = std::string((const char*)data,
                    size - sizeof(rtp_header) - extension_length);

  //assign values pointed to to an actual hard copy
    rtp_h_ = *rtp_h_p_;
    rtp_h_.seq = ntohs(rtp_h_p_->seq);
    rtp_h_.timestamp = ntohl(rtp_h_p_->timestamp);
}
```

**Figure 6.** RTP object creation algorithm.

### 2.1.6. PacketCrafter Class

The PacketCrafter class is responsible for providing functionality for the creation of SIP packets either from text files or from data the user inputs via keyboard. Furthermore, it also allows the user to send these created packets on the network by calling the send_packets function. The header file defining the PacketCrafter class is in Figure 7.

```
class PacketCrafter
{
    public:
        PacketCrafter();

        //craft the packet
        void craft_sip_packet(const std::string& sip_data,
                              uint8_t& p_num,
                              bool is_filename);

        //send packets down the stream
        void send_packets();

        bool get_user_input(uint8_t& p_num);
    private:
        std::vector<Sip> packets_;
        bool to_check, has_sdp;
};
```

**Figure 7.** Header file containing the definition of the PacketCrafter class.

#### 2.1.7. Decoding Algorithm

In order to decode the packets, we had to first eliminate the RTP packets that were being sent to the clients from the server. To do this, we used the stored IPs and ports from the previous RTP capture to create separate capture filters for each client and then started a capture using that filter to intercept the multimedia information sent to the server. Doing this would eliminate the packets originating from the server, ultimately producing an output media stream for each speaker in the captured conversation.

After splitting the RTP capture, we could continue the decoding process by pushing the packets for each user into a buffer making sure that all duplicates were eliminated, and the remaining packets were ordered by sequence number. Each packet would then be popped out of the buffer in the order they were inserted in and decoded.

The decoding process involved identifying the payload type of the packet which indicates the type of codec being used, two of the most popular being G711 µ-Law and A-Law. After the type had been identified, the packet was decoded using the created G711Codec class that acted as a wrapper for the open-source µ-Law and A-Law decoding algorithms developed by Sun Microsystems, Inc. These algorithms take an encoded value as input and output a 16-bit linear PCM decoded value. Since G711 is one of the most common VoIP codecs, it is the only one the project supports so far.

After the decoded information had been returned, it was written to a WAV (Waveform Audio File). If the decoding process is successful, the program-generated audio output files should contain the full conversation that was captured during the VoIP session. The full algorithm for the decode function can be seen in Figure 8.

```
std::string G711Codec::decode(
              uint8_t payload_type, const std::string& to_decode){

    std::string res = "";

    short out;

    for(char c : to_decode){
    //call specific decode function for payload type
    if(payload_type == CODEC_PCMA)
        out = alaw2linear((unsigned char)c);
    else
        out = ulaw2linear((unsigned char)c);

    //get memory address to out variable, cast it to a char pointer
    //and append it to the result string
    res.append((char*)&out, sizeof(out));
    }
    return res;
}
```

**Figure 8.** Full algorithm for the decode function.

### *2.2. Building an IDS Using Libtins, Apache Kafka and ksqlDB*

In this section, we discuss how we can implement an intrusion detection system by capturing network data with a custom protocol analyzer (developed in C++) and feeding it to Kafka and ksqlDB. The network traffic was captured from a virtualized environment consisting of multiple virtual machines (created and provisioned using Vagrant). Afterward, the packets were processed by the application, and the JSON (JavaScript Object Notation) output was stored in a shared folder so that it could easily be accessed by the Docker compose stack (running Kafka and ksqlDB). This approach used a modified version of the ids-ksql project [43].

#### 2.2.1. IDS Implementation Using ksqlDB

The process of creating an IDS with Apache Kafka and ksqlDB was described in detail in a blog post on the confluent.io platform [27]. The article showcases how one can process network activity in real time for intrusion detection by using ksqlDB and its SQL-like query language.

In order to emulate a live network environment, Vagrant was used. This tool allows the user to describe the virtual environment he/she wishes to create and its configuration inside a Vagrantfile. This file is then parsed when starting the system with the vagrant up command, and its contents are used by Vagrant to determine how to start and provision each virtual machine in the environment.

An outline for the virtual environment that was involved can be seen in Figure 9.

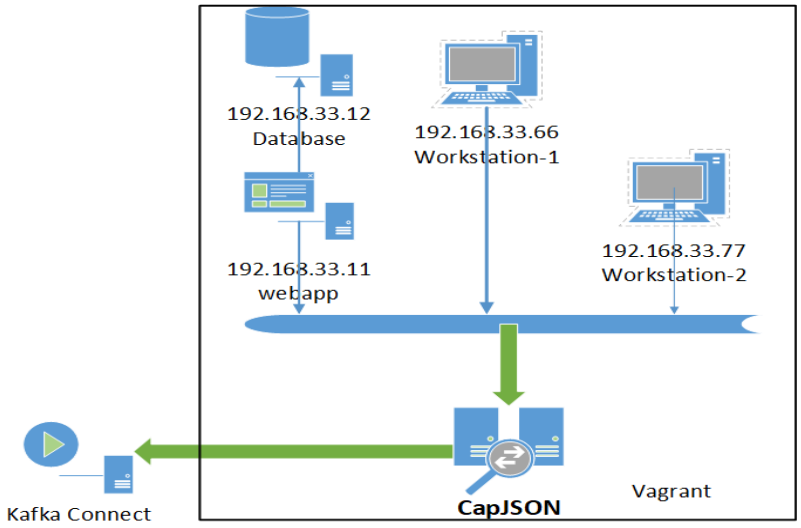


**Figure 9.** Overview of the IDS sensor and the virtual environment modified compared to [27].

It simulated a small network containing a database, a web application and two workstations that performed calls to the web application. Lastly, a network probe was used to capture and analyze all the network traffic. The implementation in [27] used tshark, a network protocol analyzer allowing to capture data from a live network and to save the outputs in different formats. For the purposes of this paper, we did not select tshark. Instead, we emulated its behavior by building a new sensor that captured all network traffic and converted it into JSON format. This was accomplished by using the libtins library. The processed packet dumps were saved to a shared folder in JSON format. Then a Kafka source connector checked its content and sent the incoming data to a Kafka cluster. The role of ksqlDB was to process and analyze the data helping to generate alerts regarding possible intrusions. Although the pipeline followed by the network packets is according to [27], note that our approach involved a different library (libtins instead of libpcap), a CapJSON application (not present in the initial solution) and a complete original package of software for a protocol analyzer available in [42–44]. This new approach allowed us to investigate the better capabilities offered by the libtins library for security of at least two major fields: SIP-based VoIP calls and large IoT networks. Coming back to the description of the pipeline, packets were firstly processed and sent to Kafka and ksqlDB using the network-traffic topic. Afterward, the raw data were structured into a network-traffic-nested topic. After that, the messages were flattened by having the network-traffic-flat topic that only kept the required fields. This topic was then analyzed to detect two types of possible attacks, potential port scan and potential slowloris. The first is a reconnaissance attack, and the second is a Denial of Service (DoS) attack. In port scanning, the hacker sends connection requests to a target device in order to test how a port responds and if it is at all active. This allows the attacker to gather knowledge about the vulnerabilities in the network and to develop a plan to infiltrate it. The second type of attack detected allows a single machine to take down another machine's web server with minimal bandwidth consumption and side effects on unrelated services and ports. Slowloris does this by opening many connections to the target server and trying to hold them active for as long as possible. This is achieved by opening connections to the target web server and sending a

partial request then periodically sending subsequent Hypertext Transfer Protocol (HTTP) headers, adding to, but never completing, the request.

#### 2.2.2. Implementing Custom Packet Capture Tool Using C/C++ and Libtins

Capturing packets and turning them into JSON objects was performed by involving a custom software protocol analyzer application developed in C++, based on libtins library, which we released to the public domain. The general structure can be seen in Figure 10. The source code can be viewed in the projects' GitHub repository [44].

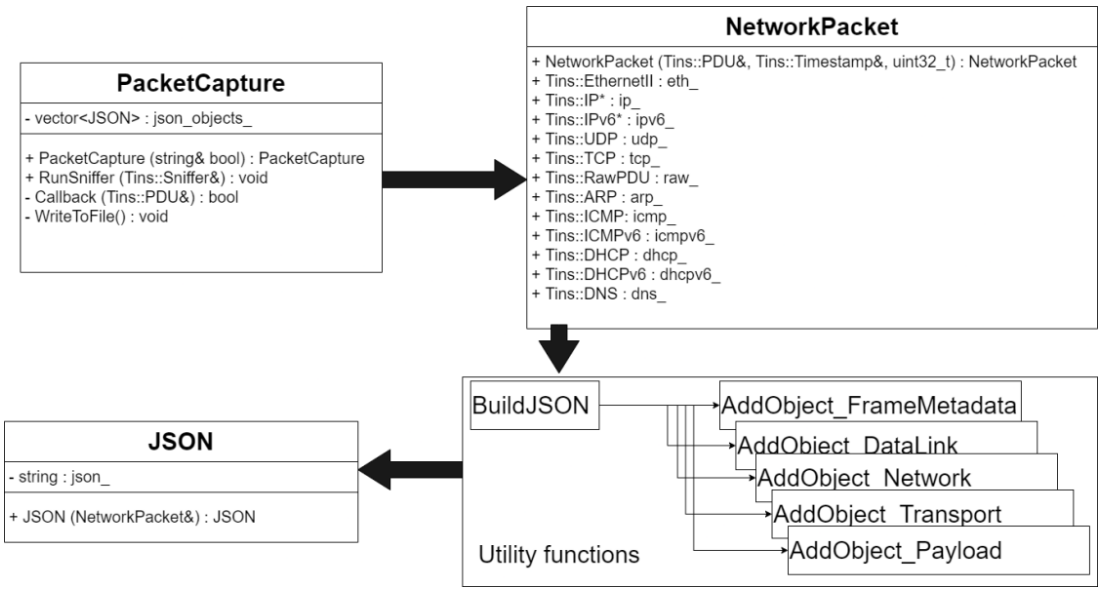


**Figure 10.** Simplified software diagram illustrating the structure of the custom sniffer application.

- PacketCapture Class

This is a simple wrapper around the packet capture functionalities of the libtins library. Under the hood, it instantiates variables that determine where the processed packet outputs will be placed. Moreover, it determines how each captured network packet is handled and processed by defining a callback function that is executed for each packet that is encountered. Inside this callback function, the raw data are structured and processed into JSON objects. Afterward, the class handles outputting the resulting objects to text files.

- NetworkPacket structure

In order to shape the raw network data captured by using the PacketCapture object, we defined a NetworkPacket structure. It mainly consists of pointers to different network protocol data types and is used to selectively initialize these protocols based on the type of packet. For example, for a UDP segment, the constructor initialized the EthernetII, IP, UDP and RawPDU pointers, splitting the captured packet into its specific layers, which were then used to process the packet.

- JSON structure

After establishing the structure of the captured network packet, a JSON object was created using the data. To achieve this, the rapidjson library was used. Note that, according to [45], this is a fast JSON parser/generator for C++, having API in both SAX/DOM styles. It seems that this has similar performance to strlen().

The processed object was stored inside a standard container that acted as a buffer. Once enough packets had been accumulated inside the buffer, the callback function from the PacketCapture class wrote the resulting JSON array to an output file.

- Utility functions

Converting the NetworkPacket to JSON using calls to the rapidjson library was done by using a group of utility functions. These involved a string buffer into which they wrote several objects consisting of key-value pairs that, when put together, made up a JSON string. The main function contains the outline for building a JSON string, and its name is BuildJSON. It first adds a timestamp, after which it proceeds to add frame metadata and layer-specific information for each captured packet. It does this by making calls to other functions that specifically handle building the JSON output for each layer (AddObject_DataLink, AddObject_Network, AddObject_Transport, AddObject_Payload). All the above functions operate very similarly, as they perform null checks on certain members of the NetworkPacket structure. Depending on the result, they determine which layers and protocols are present and which are not. Based on that, calls are made to the rapidjson library resulting in the creation of JSON objects consisting of strung together JSON key-value pairs. These calls used the inner Writer which is a Simple API for XML generator for JSON.

More examples regarding it are provided in the rapidjson library documentation [46]. To summarize, the user calls one of several functions to create a JSON object and adds key-value pairs to it, with support for adding anything specified in the JSON standard. This can be done in other ways, without using the library, but "using Writer API is even simpler than generating a JSON by ad hoc methods" [47].

#### 2.2.3. Integrating the Custom Sniffer into Ids-Ksql Project and Environment Configuration

In order to test the custom protocol analyzer and to preview the results of using the IDS, we made some changes to the original ids-ksql project. This had to be done in order to change the Vagrant configuration to include our custom network sniffer instead of tshark.

- Forking ids-ksql

The first thing to do was to fork the original GitHub repository [47] so that the changes we introduced can be version-controlled and stored in our GitHub repository [43]. This custom version of the repository was later used to run the Vagrant project with the custom protocol analyzer instead of tshark.

- Vagrant configuration

After the repository was forked, the next step was to download a local copy to our machine so that we were able to make the necessary changes to the configuration. This was done using the following command: *git clone https://github.com/adriancostin6/ids-ksql.*

With the repository cloned, the next step was to modify the Vagrantfile, located inside the infra directory, to include the configuration for creating and provisioning a custom virtual machine. Inside this, our C++ project was fetched, built and executed in order to run the custom protocol analyzer, instead of tshark. The configuration in Figure 11 instructs Vagrant to define a new virtual machine called capjson, which is the name of the custom protocol analyzer we developed.

```
config.vm.define "capjson" do |tshark|
  tshark.vm.box = "hashicorp/bionic64"
  tshark.vm.network "private_network", ip: "192.168.33.10"
  tshark.vm.hostname = "capjson"
  tshark.vm.provision "shell", path: "provision-capjson.sh"
  tshark.vm.provider "virtualbox" do |vb|
    vb.memory = "512"
    # enable promiscuous mode on the private network
    vb.customize ["modifyvm", :id, "--nicpromisc2", "allow-vms"]
  end
end
```

**Figure 11.** Vagrant configuration for creating and provisioning the capjson virtual machine.

We assigned an IP address for it and a hostname, and we ran it in promiscuous mode (important because we wanted it to process all the traffic it encounters, not just the traffic

intended for it). We configured VirtualBox to allocate 512 MB of RAM to it and selected Vagrant to use the provision-capjson.sh script in order to provision the virtual machine.

- Provisioning script

This script creates an environment in which our protocol analyzer can run and sniff network traffic. In order to do this, we had a Debian-based Linux distribution, on which we installed all the necessary prerequisites.

To use our own protocol analyzer, we needed a way to build the source code inside this virtual machine. This was done with the help of a Docker container. We chose this solution because we wanted to ensure a way to rapidly and consistently deploy our software, in a repeatable manner. We also wanted our application to be isolated and segregated from all the other applications. The configuration presented in Figure 12 shows exactly how we built the protocol analyzer using Docker.

```
#!/bin/sh

apt-get update -qq
apt-get install -qy avahi-daemon
apt-get install -qy apt-transport-https \
        ca-certificates\
        curl \
        gnupg-agent \
        software-properties-common \
        git

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
add-apt-repository -y \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
 $(lsb_release -cs) \
 stable"

apt-get install -qy docker-ce docker-ce-cli containerd.io

groupadd docker

usermod -aG docker vagrant

# get my fork of ids-ksql
git clone https://github.com/adriancostin6/ids-ksql.git

# got to my protocol analyzer project
cd ids-ksql/CapJSON

docker build -t capjson .

# move the created service for running my application to systemd
cp /vagrant/capjson.service /etc/systemd/system
systemctl daemon-reload

# start my custom protocol analyzer service
systemctl enable capjson.service
systemctl start capjson.service
```

**Figure 12.** Vagrant provisioning script for the capjson virtual machine.

After configuring the machine to properly use it, we cloned our fork of the ids-ksql repository and entered the CapJSON directory, where we stored our Docker configuration for building the project. Afterward, we ran a Docker build command to generate an image and copied the service file created for running a Docker container with our built application to the systemd directory so we could start our service with the systemctl start command. This configuration is similar to the one used in our initial project [44], with added modifications for the custom protocol analyzer application introduced in the current paper.

- Dockerfile for building the protocol analyzer

The Docker configuration for building an image can be seen in Figure 13. It used an Alpine Linux image, as this provided a simple, compact environment for us to run our code in (the size of a bare-bones Alpine container is about 8MB). On top of that image, we installed some basic dependencies for the CapJSON protocol analyzer (i.e., libpcap,

openssl, libtins and rapidjson). This was cloned from the official repository just prior to running the build [44]. The repository configuration had to be changed to include the testing mirror, as some packages were not available in the stable repositories.

```
    FROM alpine:3.10
    RUN echo
    "https://dl-cdn.alpinelinux.org/alpine/edge/testing" >>
/etc/apk/repositories

    RUN apk update -qq && apk upgrade

    RUN apk add --no-cache coreutils git build-base cmake libpcap-dev
openssl-dev libtins-dev

    RUN git clone https://github.com/adriancostin6/CapJSON.git

    RUN \
    cd CapJSON && mkdir external && cd external && \
    git clone https://github.com/Tencent/rapidjson.git && \
    cd rapidjson && mv include ../ && rm -rf * && mv ../include .

    RUN mkdir /CapJSON/build && \
    cd /CapJSON/build && cmake .. && make && mv cap-json /bin

    COPY run.sh /
    RUN chmod +x /run.sh
    VOLUME /logs
    CMD exec /run.sh
```

**Figure 13.** Dockerfile that cloned, built, and installed the C++ protocol analyzer project.

If the build process is successful, our program copies the generated binary inside the /bin directory so that it can be executed from anywhere on the system. This was handled when the Docker container was started with a simple script, which ran the application and specified the output directory for the JSON dump. See the script in Figure 14.

```
    #!/bin/sh
    exec cap-json /logs
```

**Figure 14.** Run script for executing the IDS sensor.

One of the most important facts to be mentioned is that the Dockerfile specified the creation of a volume inside the /logs directory. This was later linked to another file in the Vagrant root directory, acting as common storage between the Vagrant virtual machine environment and the Docker-composed stack running Kafka and ksqlDB. This is what allowed ksqlDB to ingest and process the captured network packets.

- Creating and configuring a service for the protocol analyzer

For running the Docker container inside the Vagrant virtual machine, a service that can be started with systemctl was created. Figure 15 is a preview of the configuration that is run when starting the service. As we can see, the service creates a data directory inside the vagrant root folder and links /data/logs to the /logs folder internal to the Docker container. It also specifies parameters for different options such as setting environment variables and network options when starting the container. This is a modified version of the tshark.service file provided in the original projects' source code on GitHub [47].

```
[Unit]
Description=Docker container for CapJSON application
BindsTo=docker.service
After=docker.service

[Service]
Environment=NAME=cap-json
Restart=on-failure
RestartSec=10
ExecStartPre=-/bin/mkdir -p /vagrant/data
ExecStartPre=-/usr/bin/docker kill ${NAME}
ExecStartPre=-/usr/bin/docker rm ${NAME}
ExecStart=/usr/bin/docker run --name cap-json \
                          -v /vagrant/data/logs:/logs \
                          -e INTERFACE=eth1 \
                          -e LINES_BY_FILE=500 \
                          --privileged \
                          --net=host \
                          capjson
ExecStop=/usr/bin/docker stop cap-json

[Install]
WantedBy=multi-user.target
```

**Figure 15.** Service configuration for starting the protocol analyzer inside the virtual machine.

## 3. Results

This section outlines the results of the research work, with the first part focusing on the VoIP analyzer command-line interface and the second one on the IDS results.

### *3.1. Results for the SIP/RTP Packet Extractor*

Using the SIP and RTP packet processing functionality described in the previous sections alongside the existing packet capture capabilities of the libtins library resulted in the creation of a command-line application. This intercepted VoIP calls and generated output files that the user can open to listen to the conversation or to visualize the SIP signaling messages. In addition, the application also supports SIP packet creation and verification, according to the RFC3261, and sending of these packets across the network. Instructions for building the command-line application and the full source code can be found inside the GitHub repository [42]. The terminal-based user interface allows the user to choose between three modes of operation. The first two deal with packet capture and processing from the default network interface or from a pcap file. The third mode of operation involves the creation of SIP signaling packets by inputting data either from a text file or the keyboard. This can be seen in Figure 16.

```
################################################################################
#                                  VoIP Analyzer                               #
################################################################################
#  To start a live packet capture: start capture                               #
#  To read packets from a pcap file: read pcap file                            #
#  To start the packet creator: packet creator                                 #
################################################################################
#  Input selected mode below                                                   #
################################################################################
start capture
Network interface: wlp3s0 IP ADDR: 192.168.1.5
Starting live capture. Press any key to stop

################################################################################
#  Generate SIP and audio output: output                                       #
#  Generate SIP output files: output sip                                       #
#  Generate audio output from RTP stream: output audio                         #
################################################################################
output
```

**Figure 16.** Text-based user interface of the VoIP Analyzer command-line application.

#### 3.1.1. Performance

Capturing and processing a VoIP call was the main function of the application. For it to be a reliable alternative solution to other similar programs, the developed command-line interface had to have comparable performance to the others. To measure this, we gathered data from three different test scenarios. This allows the reader to have some insights into the application performance (drawbacks and improvements) compared to the Wireshark packet analyzer, de facto reference for the IP-based communications community. It is for further work to extend this comparison to other solutions, such as those mentioned in [48].

#### 3.1.2. Capturing and Processing a VoIP Call

The "processing time" parameter listed in the above and following graphs is used to measure the packet processing duration. This is done in two cases, either for a certain number of packets in the individual SIP and RTP tests or for various call durations as seen in Figure 17 and Table 1. This time was measured using the real (elapsed) parameter outputted by running the time command on the application in a Linux environment.

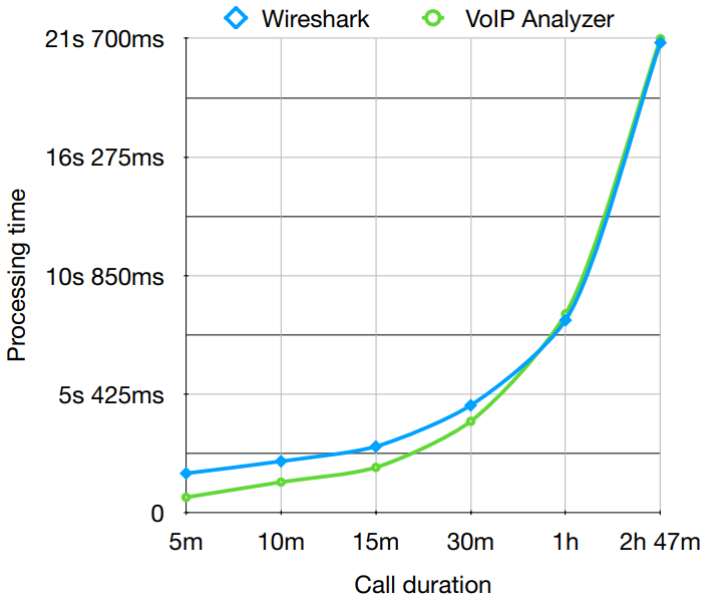


**Figure 17.** Packet processing time comparison for different VoIP call durations.

**Table 1.** Packet processing time comparison for different VoIP call durations.

| Call Information | | Processing Time (Milliseconds) | |
|---|---|---|---|
| **Duration (Minutes)** | **Number of Packets** | **Wireshark** | **VoIP Analyzer** |
| 5 | 74,250 | 1785 | 689 |
| 10 | 147,144 | 2338 | 1384 |
| 15 | 220,309 | 3015 | 2057 |
| 30 | 454,741 | 4900 | 4167 |
| 60 | 882,308 | 8800 | 9089 |
| 167 | 2,460,609 | 21,505 | 21,688 |

The first scenario that was tested was that of intercepting a Voice over IP session of variable length. To gather the data, six different captures were realized for calls ranging in length from five minutes to almost three hours. The resulting graphic and table show that if the conversation length is up to 30 to 45 min, the developed application has a lower processing time than the Wireshark packet analyzer. Afterward, when we reach longer call durations, the two packet analyzers tend to merge together in terms of performance. In the next two sections, we focus on the performance of the application when exclusively processing either SIP or RTP packets.

#### 3.1.3. Processing a Large Number of SIP Packets

The second test scenario was developed to gauge the performance of the application when handling SIP signaling packets. To do so, an instance of the program was run in order to process a large number of SIP packets, and the results were used to determine how the number of processed packets affects the execution time.

Figure 18 shows an overall performance comparison between the tool we developed and Wireshark. From it, we can see that the performance of the two tools is similar, with our implementation coming up just a few seconds short of matching Wireshark's processing speed. Both VoIP Analyzer based on libtins and Wireshark use the same packet capture engine behind the scenes, namely libpcap. Due to this, we can say that the rate at which the packets are captured is the same. The small performance difference is due to the packet processing and storing implementation. Currently, the algorithm uses the C++ STL (Standard Template Library) which adds a bit of overhead in exchange for convenience and ease of use. Another factor influencing the performance is the fact that because the application was designed to output the captured packets in the form of ".pcap" and text files, most of the computation time is spent on I/O calls. Thus, the performance gap can easily be bridged with some optimizations in the future. For capturing VoIP traffic in 5G/B5G environments, we plan to save data extracted from SIP/RTP packets directly into an Apache Kafka database. This would eliminate the need for intermediate storage and would improve the performance of the application significantly.

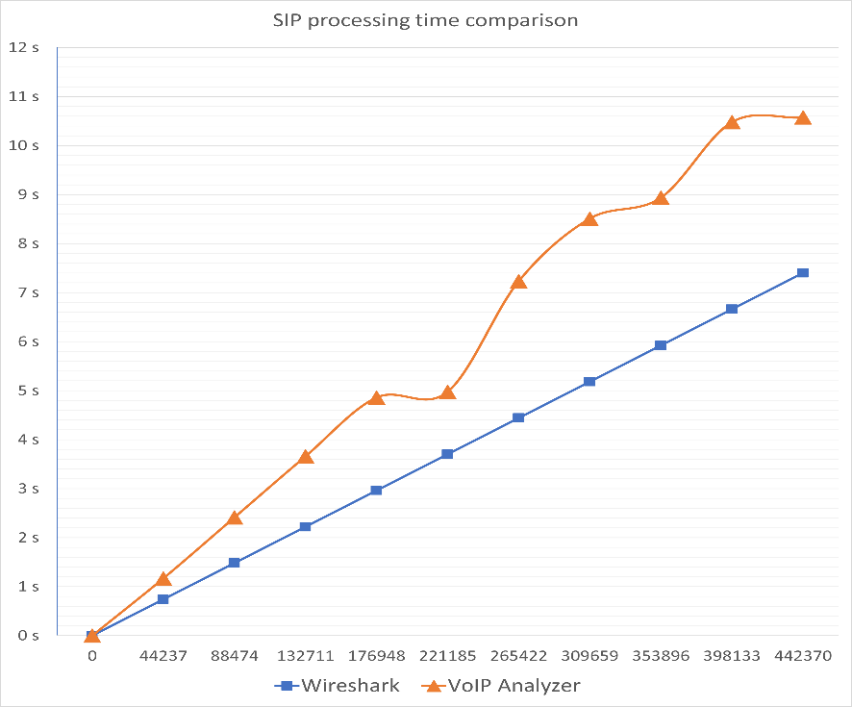


**Figure 18.** Packet processing time comparison for a varying number of SIP packets.

Furthermore, Figure 17 showcases that the SIP packet processing time is not a bottleneck in the overall performance of the application. The reasoning behind this is that during a normal phone call, the number of exchanged SIP packets was drastically lower than the number of RTP packets. For two million RTP packets that were sent during the call, only 50 to 60 SIP packets at most were exchanged.

#### 3.1.4. Processing a Large Number of RTP Packets

The third test scenario is of much similarity to the second one, involving the processing of RTP packets instead of signaling packets. Figure 19 shows a similar performance between the two compared packet analyzers.

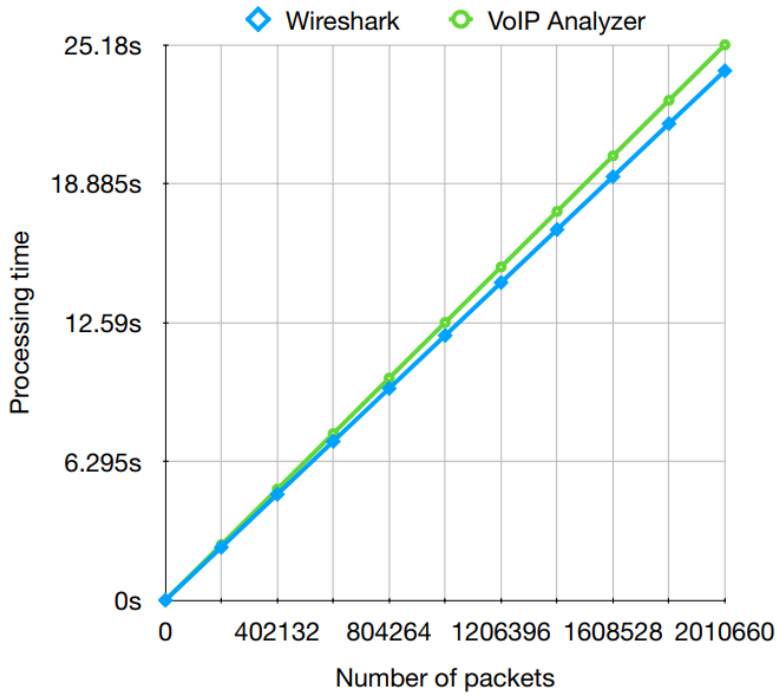


**Figure 19.** Packet processing time comparison for a varying number of RTP packets.

### *3.2. Results for the IDS*

#### 3.2.1. Running the Simulation

Once all the configurations had been done, the user started the system by running the commands shown in Figure 20. These started the Vagrant virtual machines to simulate a small network and the Docker to compose a stack containing Kafka and ksqlDB. As it was a resource-consuming process, it took up some time and failed if the user did not have enough CPU and memory to run it.

```
git clone https://github.com/adriancostin6/ids-ksql
cd ids-ksql/infra
vagrant up
cd ../ && docker-compose up -d
docker-compose exec ksql-cli ksql http://ksql-server:8088
```

**Figure 20.** Commands that set up the environment in order to test the IDS.

After the simulation environment was up and running, the user was greeted by the ksql-cli command-line interface. The next step was to specify a Network Traffic Connector that scanned the specified directory (the shared file system in this case) for incoming files, before sending them into a Kafka topic. The connector used in this project was the same as the one specified in [27]. It scanned the /data/logs directory inside the infra Vagrant project, where our sniffed packets were placed.

To detect the attacks described before, the network-traffic-nested and network-traffic-flat streams needed to be created using the ksql-cli console interface. Afterward, we created two tables for detecting the attacks using the console interface. The first table for detecting port-scan attacks can be seen in Figure 21. By counting the packets between hosts, it can trigger an alert when there are lots of packets between two hosts on several different ports. In this case, we had a threshold of 1000 datagrams for a given IP address. This meant that this IP established at least one thousand connections over the network with different hosts and involved distinct ports. Our investigations confirmed the supposition from [27] that this situation should/could be considered “suspicious”.

```
CREATE TABLE potential_port_scan_attacks
AS SELECT ip_source, COUNT_DISTINCT(ip_dest + tcp_port_dest)
FROM NETWORK_TRAFFIC_FLAT WINDOW TUMBLING (SIZE 60 SECONDS)
GROUP BY ip_source HAVING COUNT_DISTINCT(ip_dest + tcp_port_dest) >= 1000;
```

**Figure 21.** Creating a table in ksql-cli for detecting port-scan attacks.

To reproduce this attack, the user logged into the Compromised workstation in the Vagrant virtual machine network using the command: *vagrant ssh compromised*. Once he/she had access to the machine, the command *nmap -n -sT -sV -sC 192.168.33.0/24* was running to execute a nmap port scan on the 192.168.33.0/24 network (if the –p option is not specified, nmap scans the most common 1000 ports).

The slowloris attack was detected in a similar manner by creating a table in the ksql-cli that counted the number of connection resets sent by the server in the TCP segments. If there was a peak in the number of RST segments sent, then we suspected a type of DoS or Distributed Denial of Service (DDoS) attack. The structure of the detection table for the slowloris attack can be seen in Figure 22.

```
CREATE TABLE potential_slowloris_attacks
AS SELECT  ip_source, count(*) as count_conn_reset
FROM NETWORK_TRAFFIC_FLAT WINDOW TUMBLING (SIZE 60 SECONDS)
WHERE tcp_flags_ack = '1' AND tcp_flags_reset = '1'
GROUP BY ip_source HAVING count(*) > 100
```

**Figure 22.** Creating a table in the ksql-cli to detect slowloris attacks.

To simulate such an attack, the user connected to the Compromised workstation via the following command *vagrant ssh compromised*, and once he/she had access to the machine, the attack was launched by typing *slowhttptest -c 10000 -H -g -o slowhttp -i 10 -r 500 -t GET -u http://web.local:8080 -x 24 -p 3*. We want to mention again that we used the same principle for detecting the suspicious cases as in [27], but the testbed and the software packages were completely different. Maybe more than in the VoIP case, this kind of vulnerability is very likely to happen in large wireless sensor networks. Therefore, libtins deserved this particular work.

#### 3.2.2. Resulting Output in Ksql Command-Line Interface

This section provides an overview of the results gathered from testing the Intrusion Detection System by running the port scan and slowloris attacks. These tests ran on a local test machine on a Linux distribution configured as described in the previous sections. The following results are presented in a table format, with six columns and a variable number of rows. In Figure 23, we can see that by using the table for port-scan attacks, ksqlDB detected an unusual number of connections on the network, the largest being for IP address 192.168.33.66, marking it suspicious. The way we interpreted the table with the results is accurate because the IP address of the Compromised Vagrant machine we ran the attack from was indeed 192.168.33.66. The time was represented in UNIX format by default. For example, 1633804903919 represents 9 October 2021 at 18:41:43 GMT, the moment when the experiment was performed.

```
ksql> SELECT * FROM potential_port_scan_attacks EMIT CHANGES;
+--------------+--------------+-----------------+--------------+-----------------+-----------+
|ROWTIME       |ROWKEY        |WINDOWSTART      |WINDOWEND     |IP_SOURCE        |KSQL_COL_1 |
+--------------+--------------+-----------------+--------------+-----------------+-----------+
|1633804903919 |192.168.33.1  |1633804860000    |1633804920000 |192.168.33.11    | 1719      |
|1633804997612 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 1051      |
|1633804997688 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 1726      |
|1633804997748 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 2267      |
|1633804997837 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 2986      |
|1633804997914 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 3723      |
|1633804997995 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 4501      |
|1633805009473 |192.168.33.66 |1633804980000    |1633805040000 |192.168.33.66    | 4798      |
```

**Figure 23.** Detection of potential port-scan attacks in the ksql command-line interface.

The results for the detection of the second attack can be seen in Figure 24. This shows that the number of connection resets detected for IP address 192.168.33.11 (web server) was high, which led us to suspect that there might be some type of Denial-of-Service attack being run on the web server. This was, in fact, correct, as we explained in the previous section, because during this test, we intentionally ran a slowloris attack on the web server from the Compromised Vagrant machine.

```
ksql> SELECT * FROM potential_slowloris_attacks EMIT CHANGES;
+---------------+----------------+--------------+--------------+--------------+-----------------------+
|ROWTIME        |ROWKEY          |WINDOWSTART   |WINDOWEND     |IP_SOURCE     |COUNT_CONNECTION_RESET |
+---------------+----------------+--------------+--------------+--------------+-----------------------+
|1633804873189  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  231                  |
|1633804875249  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  341                  |
|1633804877293  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  464                  |
|1633804879331  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  629                  |
|1633804879341  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  632                  |
|1633804893651  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  706                  |
|1633804895693  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 |  874                  |
|1633804897766  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 | 1041                  |
|1633804899800  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 | 1213                  |
|1633804901869  |192.168.33.11   |1633804860000 |1633804920000 |192.168.33.11 | 1451                  |
```

**Figure 24.** Detection of potential slowloris attacks in the ksql command-line interface.

## 4. Conclusions

This paper presented two use cases involving the open source libtins library. We used this instead of a well-known alternative such as libpcap because we wanted to investigate it in real-time applications in larger IoT networks and in WLANs in general. The first part of the paper included a method to extend this library with SIP and RTP classes in order to build a packet-based VoIP analyzer. The second part presented a method to have an Intrusion Detection System by creating a C++ application to act as a sniffer. The solution was based on Apache Kafka and ksqlDB, the simulations being executed within a Vagrant virtualized network. Furthermore, the basic C++ application developed by us, by capturing and processing data, generated outputs in JSON format. This allows replacing other solutions (such as tshark) and detecting port-scan and slowloris attacks. Because of its minimalist nature, the C++ tool can be extended to parse the protocols that are needed to detect other different types of attacks.

By using the libtins library, the custom software packages presented in this paper and published in GitHub proved that this tool could be used for the development of IP-based applications in general (not necessarily limited to the previously mentioned cases). The solution offers similar performance as the one provided by Wireshark, but it is for further work to compare it with other sniffers available on the market. We are aware that there is still a lot of work to prove the better performance of libtins-based sniffers mainly for wireless networks, but the results obtained herein encourage us to continue this approach in the near future. We intend to continue in the direction of capturing VoIP traffic in 5G/B5G networks for analyzing it from a security perspective.

**Author Contributions:** Conceptualization, D.Z.; Software, A.-T.C.; writing—original draft preparation, A.-T.C.; writing—review and editing, A.-T.C., D.Z. and V.D.; supervision, D.Z. and V.D. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding. The APC was funded by the Technical University of Cluj-Napoca.

**Data Availability Statement:** The data that support the findings of this study are available from the corresponding author, D.Z., upon reasonable request.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Fontanini, M. Libtins (Version 4.2) [Source Code]. Available online: https://github.com/mfontanini/libtins (accessed on 5 September 2021).
2. Libtins Documentation. Available online: https://libtins.github.io/tutorial/ (accessed on 5 September 2021).
3. Sadrhaghighi, S.; Dolati, M.; Ghaderi, M.; Khonsari, A. SoftTap: A Software-Defined TAP via Switch-Based Traffic Mirroring. In Proceedings of the 2021 IEEE 7th International Conference on Network Softwarization (NetSoft), Tokyo, Japan, 28 June–2 July 2021; pp. 303–311. [CrossRef]
4. Libpcap. Available online: https://www.tcpdump.org/ (accessed on 5 September 2021).
5. Li, J.; Wu, C.; Ye, J.; Ding, J.; Fu, Q.; Huang, J. The Comparison and Verification of Some Efficient Packet Capture and Processing Technologies. In Proceedings of the 2019 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Fukuoka, Japan, 5–8 August 2019; pp. 967–973. [CrossRef]
6. Bonelli, N.; Giordano, S.; Procissi, G. Enabling packet fan-out in the libpcap library for parallel traffic processing. In Proceedings of the 2017 Network Traffic Measurement and Analysis Conference (TMA), Dublin, Ireland, 21–23 June 2017; pp. 1–9. [CrossRef]
7. Vormayr, G.; Fabini, J.; Zseby, T. Why are My Flows Different? A Tutorial on Flow Exporters. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 2064–2103. [CrossRef]
8. Ivoševia, M.; Vranješ, M.; Pekoviű, V.; Kaprocki, Z. Client-side solution for QoS measurement of video content delivery over IP networks. In Proceedings of the 2018 IEEE 8th International Conference on Consumer Electronics—Berlin (ICCE-Berlin), Berlin, Germany, 2–5 September 2018; pp. 1–6. [CrossRef]
9. Zubow, A.; Zehl, S.; Wolisz, A. BIGAP—Seamless handover in high performance enterprise IEEE 802.11 networks. In Proceedings of the NOMS 2016—2016 IEEE/IFIP Network Operations and Management Symposium, Istanbul, Turkey, 25–29 April 2016; pp. 445–453. [CrossRef]
10. Morrell, C.; Ransbottom, J.S.; Marchany, R.; Tront, J.G. Scaling IPv6 address bindings in support of a moving target defense. In Proceedings of the 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014), London, UK, 8–10 December 2014; pp. 440–445. [CrossRef]
11. Ghazanfar, S.; Hussain, F.; Rehman, A.U.; Fayyaz, U.U.; Shahzad, F.; Shah, G.A. IoT-Flock: An Open-source Framework for IoT Traffic Generation. In Proceedings of the 2020 International Conference on Emerging Trends in Smart Technologies (ICETST), Karachi, Pakistan, 26–27 March 2020; pp. 1–6. [CrossRef]
12. Costin, A.-T.; Zinca, D. Extending the libtins library with SIP and RTP classes. In Proceedings of the 2020 International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, 5–6 November 2020; pp. 1–4. [CrossRef]
13. Gruber, M.; Fankhauser, F.; Taber, S.; Schanes, C.; Grechenig, T. Trapping and analyzing malicious VoIP traffic using a honeynet approach. In Proceedings of the 2011 International Conference for Internet Technology and Secured Transactions, Abu Dhabi, United Arab Emirates, 11–14 December 2011; pp. 442–447.
14. Aziz, A.; Hoffstadt, D.; Rathgeb, E.; Dreibholz, T. A distributed infrastructure to analyse SIP attacks in the Internet. In Proceedings of the 2014 IFIP Networking Conference, Trondheim, Norway, 2–4 June 2014; pp. 1–9. [CrossRef]
15. Wireshark. Available online: https://www.wireshark.org/ (accessed on 5 September 2021).
16. Chappell, L. *Wireshark Network Analysis*, 2nd ed.; Chappell University: Reno, Nevada, 2019; ISBN 978-1-893939-94-3.
17. Barry, M.A.; Tamgno, J.K.; Lishou, C.; Cissé, M.B. QoS impact on multimedia traffic load (IPTV, RoIP, VoIP) in best effort mode. In Proceedings of the 2018 20th International Conference on Advanced Communication Technology (ICACT), Chuncheon, Korea, 11–14 February 2018; pp. 694–700. [CrossRef]
18. Pathania, N.; Singh, R.; Malik, A. Comparative Study of Audio and Video Chat Application Over the Internet. In Proceedings of the 2018 International Conference on Intelligent Circuits and Systems (ICICS), Phagwara, India, 19–20 April 2018; pp. 251–257. [CrossRef]
19. François, J.; State, R.; Engel, T.; Festor, O. Digital forensics in VoIP networks. In Proceedings of the 2010 IEEE International Workshop on Information Forensics and Security, Seattle, WA, USA, 12–15 December 2010; pp. 1–6. [CrossRef]

20. Langthasa, B.; Acharya, B.; Sarmah, S. Classification of network traffic in LAN. In Proceedings of the 2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV), Shillong, India, 29–30 January 2015; pp. 92–99. [CrossRef]
21. RFC 3261—SIP: Session Initiation Protocol. Available online: https://tools.ietf.org/html/rfc3261 (accessed on 5 September 2021).
22. Carvajal, L.; Chen, L.; Varol, C.; Rawat, D. Detecting unprotected SIP-based voice over IP traffic. In Proceedings of the 2016 4th International Symposium on Digital Forensic and Security (ISDFS), Little Rock, AR, USA, 25–27 April 2016; pp. 44–48. [CrossRef]
23. Moon, S. Packet Sniffer Code in C using Linux Sockets (BSD)-Part 2. 2020. Available online: https://www.binarytides.com/packet-sniffer-code-in-c-using-linux-sockets-bsd-part-2/ (accessed on 5 September 2021).
24. Herculea, M.; Blaga, T.; Dobrota, V. Evaluation of Security and Countermeasures for a SIP-based VoIP Architecture. In Proceedings of the 7th RoEduNet International Conference "Networking in Education and Research", Cluj-Napoca, Romania, 28–30 August 2008; pp. 34–39, ISBN 978-973-662-393-6.
25. RFC 4566—SDP: Session Description Protocol. Available online: https://tools.ietf.org/html/rfc4566 (accessed on 5 September 2021).
26. RFC 3550—RTP: A Transport Protocol for Real-Time Applications. Available online: https://tools.ietf.org/html/rfc3550 (accessed on 5 September 2021).
27. De Bernonville, G.D.; Ribera, M. Intrusion Detection with ksqlDB. Available online: https://www.confluent.io/blog/build-a-intrusion-detection-using-ksqldb (accessed on 5 September 2021).
28. Vagrant. Available online: https://www.vagrantup.com/intro (accessed on 5 September 2021).
29. Apache Kafka. Available online: https://kafka.apache.org/ (accessed on 5 September 2021).
30. Seymour, M. *Mastering Kafka Streams and ksqlDB*; O'Reilly Media Inc.: Sebastopol, CA, USA, 2021.
31. Vyas, S.; Tyagi, R.K.; Jain, C.; Sahu, S. Literature Review: A Comparative Study of Real Time Streaming Technologies and Apache Kafka. In Proceedings of the 2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT), Sonepat, India, 3 July 2021; pp. 146–153. [CrossRef]
32. Van Dongen, G.; Van den Poel, D. Evaluation of Stream Processing Frameworks. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 1845–1858. [CrossRef]
33. Nguyen, S.; Salcic, Z.; Zhang, X.; Bisht, A. A Low-Cost Two-Tier Fog Computing Testbed for Streaming IoT-Based Applications. *IEEE Internet Things J.* **2021**, *8*, 6928–6939. [CrossRef]
34. Chen, C.; Cai, J.; Ren, N.; Cheng, X. Design and Implementation of Multi-tenant Vehicle Monitoring Architecture Based on Microservices and Spark Streaming. In Proceedings of the 2020 International Conference on Communications, Information System and Computer Engineering (CISCE), Kuala Lumpur, Malaysia, 3–5 July 2020; pp. 169–172. [CrossRef]
35. Tidjon, L.N.; Frappier, M.; Mammar, A. Intrusion Detection Systems: A Cross-Domain Overview. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 3639–3681. [CrossRef]
36. ksqlDB. Available online: https://ksqldb.io/overview.html (accessed on 5 September 2021).
37. Rocha, A.D.; Freitas, N.; Alemão, D.; Guedes, M.; Martins, R.; Barata, J. Event-Driven Interoperable Manufacturing Ecosystem for Energy Consumption Monitoring. *Energies* **2021**, *14*, 3620. [CrossRef]
38. Chira, C.-M.; Portase, R.; Tolas, R.; Lemnaru, C.; Potolea, R. A System for Managing and Processing Industrial Sensor Data: SMS. In Proceedings of the 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, 3–5 September 2020; pp. 213–220. [CrossRef]
39. Nokia. Available online: https://www.nokia.com/networks/solutions/voice-over-5g-vo5g-core/ (accessed on 30 November 2021).
40. Di Mauro, M.; Liotta, A. An Experimental Evaluation and Characterization of VoIP Over an LTE-A Network. *IEEE Trans. Netw. Serv. Manag.* **2020**, *17*, 1626–1639. [CrossRef]
41. Biondi, P.; Bognanni, S.; Bella, G. VoIP Can Still Be Exploited—Badly. In Proceedings of the 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), Paris, France, 20–23 April 2020; pp. 237–243. [CrossRef]
42. Costin, A.-T. VoIP-Analyzer [Source Code]. Available online: https://github.com/adriancostin6/VoIP-Analyzer (accessed on 5 September 2021).
43. Costin, A.-T. Ids-Ksql (Fork) [Source Code]. Available online: https://github.com/adriancostin6/ids-ksql (accessed on 5 September 2021).
44. Costin, A.-T. CapJSON [Source Code]. Available online: https://github.com/adriancostin6/CapJSON (accessed on 5 September 2021).
45. Tencent, Rapidjson [Source Code]. Available online: https://github.com/Tencent/rapidjson (accessed on 5 September 2021).
46. rapidjson Documentation. Available online: https://rapidjson.org/index.html (accessed on 9 September 2021).
47. Zenika, Ids-Ksql [Source Code]. Available online: https://github.com/Zenika/ids-ksql (accessed on 5 September 2021).
48. Watson, J. Eleven Best Packet Sniffers in 2021, Comparitech Limited. 2021. Available online: https://www.comparitech.com/net-admin/packet-sniffer-network-analyzers/ (accessed on 9 October 2021).